Digital Circuit Design

Springer-Verlag Berlin Heidelberg GmbH

N. Wirth

# Digital Circuit Design for Computer Science Students

## An Introductory Textbook

With 147 Figures
and 1 Fold-out Diagram

Springer

Prof. Dr. Niklaus Wirth
Institut für Computersysteme
ETH Zürich
CH-8092 Zürich

ISBN 978-3-540-58577-0

Library of Congress Cataloging-in-Publication Data

Wirth, Niklaus: Digital circuit design for computer science students: an introductory textbook / Niklaus Wirth
p. cm. Includes bibliographical references and index.
ISBN 978-3-540-58577-0 ISBN 978-3-642-57780-2 (eBook)
DOI 10.1007/978-3-642-57780-2

1. Electronical digital computers-Circuits 2. Digital circuits-design and construction. I. Title
TK7888.4.W57 1995 621.39-dc20 95-7220 CIP


Originally published by Springer-Verlag Berlin Heidelberg New York in 1995



Typesetting: Data Conversion by Text & Graphik GmbH, Heidelberg. Cover design: Künkel + Lopka, Ilvesheim. Production Editor: P. Strasser, Heidelberg.

SPIN 10484890 33/3142-5 4 3 2 1 0 - Printed on acid-free paper

# Preface

This book emerged from lecture notes of a course taught in the second year to students of Computer Science at the Federal Institute of Technology, Zürich.

The topic of hardware design plays a relatively minor role in Computer Science curricula at many universities. Most courses concentrate on the various aspects of theory, software, and of information systems. Students therefore obtain few opportunities to deal with concrete engineering problems and physical devices. We consider this as rather unfortunate, particularly for technical universities. As a result, we observe a growing gap between interest in and understanding of design issues involving not only software but also hardware and interfaces. This is regrettable at a time when new and advanced solutions to many problems are often crucially influenced by recent hardware developments, at a time when the engineer needs to be competent in both software and hardware issues in order to find an optimally integrated, competitive solution.

It turns out that the hesitation of many students in Computer Science to take an active interest in hardware - his or her daily tool! - does not only stem from a preference of "clean", abstract concepts with a corresponding distaste for dealing with concrete components, construction techniques, and the "dirty" realities inflicted by nature, but also stems from the lack of a bridge between the two realms. On the one hand, courses on computer architecture find their climax in presenting and comparing systems by highlighting their complex data paths between register banks, typically without mentioning difficulties and cost of implementations. On the other hand, courses on digital design end by presenting finite state machines, methodologies for constructing them, and recipes for optimizing them with nauseating details. The bridge leading from state machines to programmable computers is missing. Hence, the relevance of knowledge about digital circuit design remains obscure to many students of Computer Science.

The primary objective of this text is therefore to close this gap. The book culminates in the development of a simple, yet complete computer in Chapter 8. Its functioning and its circuits are explained in full

detail, including the interpretation cycle of instructions and all control signals activating the various units along the data paths. By simple is meant that the design be devoid of facilities that are inessential and would only contribute to obscure the essentials. By complete is meant that the computer features all concepts necessary to be programmable and flexible. To nobody's surprise, it therefore directly reflects the basic computer structure postulated in 1945 by John von Neumann, upon which all subsequent developments have been based. An implementation of our design, cynically named Hercules, using standard electronic parts, concludes this central chapter.

The preceding chapters present the necessary ingredients. Chapter 2 is devoted to combinational circuits. It starts out with the basic Boolean operators and their representation in the form of gates. Some frequently encountered combination patterns are introduced: the decoder, multiplexer, adder, and multiplier. The subject of expression simplification is treated lightly and almost in passing. This is not only because in modern technologies the number of gates in a circuit is no longer the only measure for its effectiveness or cost, but also since the task of simplification is increasingly left to computerized tools. Then, normal forms of expressions are introduced, leading to their realizations in terms of 2-level circuits, optimally suited for programmable incarnations, the ROM, PLA, PAL, and FPGA.

Chapter 3 introduces registers. The starting point is the SR-latch which, in distinction to combinational circuits, introduces the feedback loop. Another small addition yields the transparent D-latch, from which the edge-triggered D-register is readily derived.

Combinational circuits and registers together form sequential circuits. They are the topic of Chapter 4. Restricting attention to synchronous circuits, the finite state machine is discussed. Typical examples include the shift register and counters. The chapter's main part presents a recipe for deriving a state machine corresponding to an arbitrary, algorithmic specification. It ends with a presentation of programmable devices (PLDs) with registers, as they are today's primary choice for implementing state machines.

Chapter 5 introduces the concept of the bus to reduce the number of interconnections between units of a digital system. The two prevalent

techniques of the open-collector bus and the tri-state bus are explained. It is shown that the open-collector - actually open-drain - scheme is fundamental in the realization of highly regular structures such as ROMs, PALs, and PLDs.

Memories are the subject of Chapter 6, explaining their basic cells and the structures in which cells are arranged. The treatment of static and dynamic RAMs is admittedly rather cursory. Insight and understanding are our principal objectives here, rather than specific know-how needed for their design. Knowledge about the matrix structure, for example, lets the reader recognise the simplicity and usefulness of the VRAM structure, which is used to greatest advantage for display frame buffers in all modern computers.

The building elements accumulated so far suffice to construct an entire computer. Before this formidable task is tackled, however, the book turns in its Chapter 7 to a subject crucial in the design process of any circuit of nontrivial complexity: the notation used for its specification. A language called Lola is introduced. It is sufficiently powerful to express any sequential circuit in a succint and structured fashion, and it is sufficiently simple to be described in its entirety in a few pages only. It is not artificially inflated by a multitude of embellishments and "convenient" abbreviations, nor by annotations and hints directed at implementation or simulation tools. It concentrates on its single purpose, to describe circuits, i.e. static objects. This contrasts with programming languages describing dynamic processes. A process is executed, a circuit is not. Despite this difference, the appearance of our notation, i.e. its syntax, owes much to programming languages. And this is intentional in view of the expected readership. Declarations introducing objects (variables) with names and specifying their structure, and statements assigning an expression to - or better: defining the value of - a variable, are both familiar concepts to programmers.

Of particular importance is the facility to define new types of circuits and to instantiate them. This mirrors the fact that mostly circuits consist of multiple instances of identical subcircuits. Their inputs and outputs appear as parameters. Furthermore it is possible to supply an instantiation with numeric parameters which typically determine ar-

ray dimensions. Hence, such a type declaration stands for a multitude of identical circuit patterns. For example, a single declaration suffices to describe all binary counters, no matter how many digits they represent. This kind of flexible parametrization is indeed the principal advantage of textual notations over symbolic schematics. The textual notation can readily be processed by software tools to aid in the implementation of the specified circuit, that is, in determining a layout in terms of the available devices and technology.

After all these preparations, the ground has been laid for designing the mentioned Hercules computer. It should be noted that the level of abstractions so far used is that of the purely digital circuit with two signal values 0 and 1. Only on a few occasions electrical properties are mentioned, namely where ignoring them would be unrealistic. As prerequisites for this course we postulate a basic knowledge about electricity and electronic devices. The notions of charge, current, voltage, resistance and capacitance should be understood, as well as the basic properties of active elements such as diodes and transistors. Chapter 1 is a brief summary of what is indispensible for the understanding of digital gates. It explains how gates are constructed from bipolar and field effect transistors.

The remaining chapters are devoted to various topics. They all depend on the material presented earlier, but are mutually independent. Hence they can freely be selected or omitted in an introductory course, depending on the time available.

Chapter 9 pursues two diverse aims. On the one hand, it elaborates on two short algorithms and their formulation in terms of Hercules instructions. Thus, they provide a demonstration of the usefulness of its architecture. The algorithms are those for multiplication and division based on repeated addition or subtraction and on shifting. The repetitions are those of a fixed multiply or divide step. On the other hand, the chapter presents equivalent hardware solutions of the two steps. Thereby, the notion is conveyed that solutions to specific software tasks can be supported, simplified, and made more efficient by adding appropriate hardware.

Chapter 10 explains the design of a small computer system based on modern, highly integrated components, in particular on a 32-bit mi-

croprocessor. Again, we concentrate on the truly indispensible parts of a system, but nevertheless perform the essential step from a circuit primarily geared towards the needs of a tutorial to a system that can be used for genuine applications in today's practice.

So far, the text had been restricted to the design of synchronous circuits, and we emphasize the advice that engineers will greatly benefit by adhering to this restriction as well whenever possible. A case where it is impossible is the connection of larger (in themselves synchronous) units of a system for exchanging data among them. Chapter 11 discusses ways to connect units that are asynchronous with respect to each other. It introduces the notion of protocol, and in particular the so-called handshake. An example of a standard making use of it is the SCSI bus. Then follows a presentation of a simplified version of this scheme, which is typically used for interfacing external devices with a computer bus. This is illustrated by the example of adding an input and an output port to the Hercules computer built in Chapter 8.

Following the subject of parallel interfaces, it appears mandatory to also treat the topic of serial data transmission. Chapter 12 starts out by presenting a simple transmitter and receiver operating on the same clock, transmitting data in small packets. This technique is used in many embedded applications and has found its way into industrial standards. The technique of synchronous transmission by encoding data and clock signals onto a single wire is then discussed. A more economical use of the available bandwidth, however, is obtained by using a separate clock at the receiver. This leads to asynchronous transmission. By dropping the requirement of a common clock, we arrive at the circuit commonly known as the universal, asynchronous receiver and transmitter (UART). Finally, the design of a buffered transmitter and receiver pair is presented which achieves a transmission rate of 10 Mb/s.

Every course on a technical subject like ours must be accompanied by exercises to reinforce what had been learnt and to let the student discover lacks of understanding. We know that exercises performed on paper only are insufficient. Exercises in a laboratory involving physical parts contribute significantly to the motivation of engineering students. The traditional digital laboratory provides building elements

containing basic circuits that can be plugged, wired, or soldered together. This inherently costly setup usually limits the number of components available, and thereby restricts exercises to relatively trivial problems. Fortunately, modern semiconductor technology paved the way to a much more satisfactory laboratory: the entire collection of building blocks is replaced by a single, field-programmable gate array (FPGA). This device, available in the form of a single chip, incorporates a matrix of identical cells, typically capable of representing a number of simple, combinational circuits, a register, or a combination thereof. Cells, in particular neighbouring cells, can be connected. A cell's function and its connections are determined by gates (switches) whose state (open/closed) depends on values held by latches. These latches form a static RAM. Hence, a circuit is implemented by loading the underlying RAM with data generated by an appropriate circuit layout editor, typically based on a graphical view of the cell array on a display. It is obvious that a laboratory based on FPGAs is not only much less costly, but also permits the implementation of much more realistic, interesting, and challenging exercises. After all, modern FPGAs contain up to several thousand cells.

Exercises are supplied as is customary for textbooks. We concentrate on design exercises. For larger selections of smaller and theoretical problems we refer to the literature listed below. Appendix 1 contains a description of an FPGA architecture, and implementations of several of the circuits presented in the text are given in the form of layouts for this FPGA architecture. The solutions include designs of the Hercules computer and of a UART.

These solutions are also available in electronic form, together with the software required for editing, analyzing, and checking them. This software includes a compiler for the language Lola, together with a tool called the Checker. The latter verifies whether or not a layout is consistent with its specification in terms of Lola. A PC-extension board is available containing the mentioned FPGA used by the described software tools.

Zürich, January 1995

N. Wirth

# Table of Contents

Further Reading

J. P. Hayes. Digital Logic Design. Addison Wesley, 1993

R. Katz. Contemporary Logic Design. Benjamin/Cummings, 1994.

E. J. McCluskey. Logic Design Principles. Prentice Hall, 1986.

J. D. Nicoud. Microprocessor Interface Design. Chapman & Hall, 1991

# Transistors and Gates

1.

**Overview**

The elementary components of digital circuits are gates. They are composed of transistors as electrically active elements. Here we review those properties of transistors which are relevant for the understanding of their function within gates. The two most frequently used kinds of transistors are the bipolar and the field effect transistors. The former are the basis of transistor-transistor logic (TTL), the latter of metal-oxide-semiconductor technology, which is predominant in highly integrated components. Although the behaviour of the two types of transistors is essentially the same, they differ in certain properties.

## 1.1 Gates with Bipolar Transistors

Apart from passive elements such as resistors and capacitors, circuits consist of active elements which amplify a current or a voltage. The ubiquitous active element is the *transistor*. For the time being we regard it as a black box as shown in Fig. 1.1, and consider its transfer characteristic, i.e. its output voltage as a function of its input voltage.

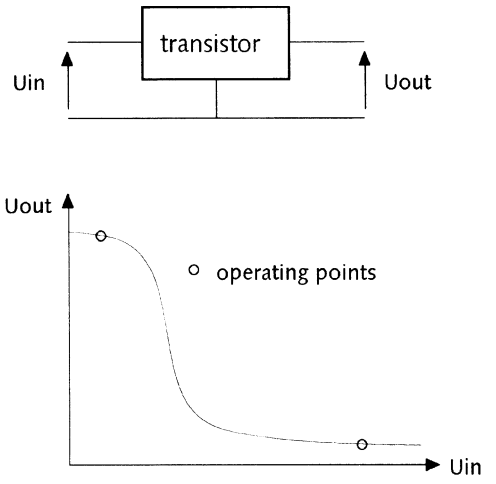



*Fig. 1.1 Transistor characteristic*

*Linear circuits*

*Digital circuits*

Whereas in so-called *analog,* or *linear circuits,* signals have a continuous range of values, and transistors operate in the linear part of their transfer characteristic, in so-called *digital circuits* only two distinct signal values occur, and transistors operate in one of two possible points of their characteristic. These points are marked in Fig. 1.1 by two small circles. The distinguishing feature of these operating points is that the output hardly reflects small changes of the input, since they lie in almost horizontal parts of the characteristic. This implies that the transistor has a high insensitivity against small input signal changes possibly caused by various ephemeral effects; this property is called *noise immunity,* and is an absolute requirement for digital circuits, where millions of transistors may have to cooperate and reliability is of paramount importance. In fact, it is the reason for the success of digital circuits in competition against analog circuits. The two operating points are attributed the "logical values" 0 and 1.

*Gate*

*Bipolar transistor*

*Emitter, base, collector*

A circuit representing an elementary logical function is called a *gate.* Implementations of gates depend at least to some degree on the technology used. We first consider the so-called *bipolar technology.* A *bipolar transistor* essentially is a current amplifier and consists of three layers of semiconductor material as shown in Fig. 1.2. The layers are called *emitter, base,* and *collector,* suggesting that current flows from the emitter to the collector and is controlled by the intermediate layer, the base. We may regard the emitter-collector path as two diodes which bar any current flow. However, if a current, i.e. charge carriers, are injected at the base, the emitter-collector path becomes conducting. In digital circuits, either no current is injected, causing the transistor to be non-conducting, or a reasonably large current is injected, causing the transistor to be saturated, in which case the voltage between collector and emitter is near zero.

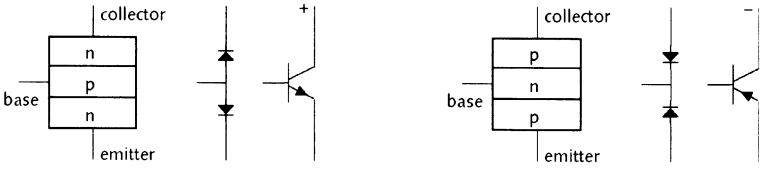



*Fig. 1.2. Bipolar n-p-n and p-n-p transistors*

Depending on the electrical properties of the three layers caused by their doping, transistors with current flowing in either direction can be created. n-p-n transistors use a positive source voltage. Majority charge carriers (electrons) flow from the emitter to the collector, i.e. the current flows - by convention - from the collector to the emitter (despite of their names). p-n-p transistors use a negative source voltage. Current flows from the emitter to the collector.

*n-p-n transistors*

*p-n-p transistors*

The simplest gate is the inverter; it consists of a single transistor and a resistor. The basic circuit is shown in Fig. 1.3. Also shown is the transistor's operating characteristic. We remember: if the input voltage is low (near zero), the transistor is non-conducting, and therefore the output voltage is (almost) equal to the supply voltage. If the input voltage is high, the transistor saturates and acts as (almost) a short circuit, and the output voltage is low.

*Inverter*

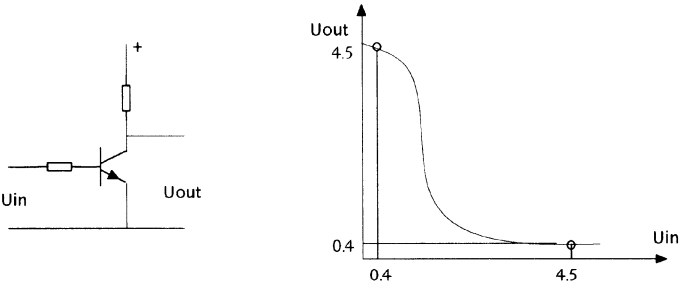



*Fig. 1.3. Inverter circuit and operating points*

Apart from the inverter, the so-called Nand gate is also fundamental. It can be regarded as a generalized inverter with two (or more) inputs. Its output is low, if both (all) inputs are high, and high otherwise. It is implemented like the inverter, but with the addition of a multi-emitter transistor in front (see Fig. 1.4). Its operation can be explained as follows:

*Nand gate*

*Multi-emitter transistor*

If at least one input is set to low, a base current flows through T1 and R1. Thereby the collector of T1 obtains almost the same voltage (potential) as its emitter, and therefore no base current flows through T2, causing the output to be high. If all inputs are high, T1's gate-collector diode becomes conducting, drawing a small base current from T2, which saturates and causes the output voltage to be low.

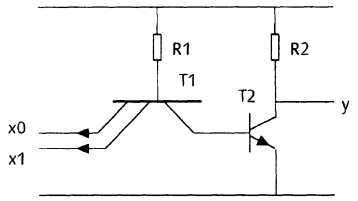



*Fig. 1.4. Nand-circuit using a multi-emitter transistor*

Bipolar gate circuits usually add an output stage in order to increase their driving power. In particular, the resistor R2 of Fig. 1.4 is replaced by another transistor, making the output stage symmetric. This is shown in Fig. 1.5. Since transistor T4 appears to sit on top of T3, this arrangement is called a *totem-pole* output stage. If any input x is low, T2 and T3 are non-conducting, and hence current flows from the base of T4 through R2, saturating T4 and pulling the output high. If no input is low, T1 is non-conducting, but a small current flows through R1 from the base of T2, making T2 conducting and saturating T3, yielding a collector voltage near zero. The collector voltage of T2 is about 1 volt, which is too little above the emitter voltage of T4 (which is equal to the collector voltage of T3 plus the voltage drop across the diode) to make T4 conducting. Therefore, either T3 is conducting or T4, but never both. However, when T2 switches to the non-conducting state, T4 may become conducting briefly before T3 turns off, causing a momentary short circuit between the supply and ground. In order to limit the resulting current spikes, a small resistor R4 (about 100 Ω) is put in series with T4 and T3.

*Totem-pole output*

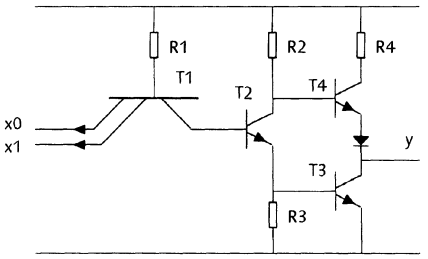



*Fig. 1.5. Nand-circuit with totem-pole output stage*

## 1.2. Gates with Field Effect Transistors

In field effect transistors (FET) the conductivity of the area between emitter and collector is controlled not by injection of charge carriers, i.e. by a base current, as in bipolar transistors, but by applying an electric field. A FET has the structure as shown in Fig. 1.6. The term emitter is replaced by *source*, base by *gate*, and collector by *drain*.

*Source, gate, drain.*

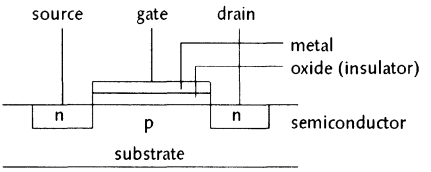



*Fig. 1.6. Cross-section of field effect transistor*

If no voltage is applied to the gate with respect to the substrate (and the source), the area under the gate is devoid of charge carriers, i.e. is not conducting. If a (positive) voltage is applied to the gate, carriers are induced in the area; the gate and the p-doped area, together with the insulator in between, form a capacitor. The charge carriers make the channel between source and drain conductive. The symbol for a FET and its characteristic are displayed in Fig. 1.7.

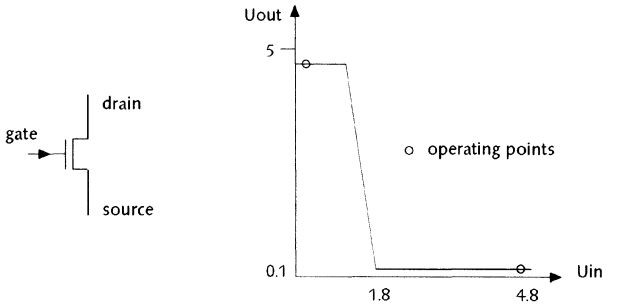



*Fig. 1.7. Symbol and characteristic of field effect transistor*

Circuits with FETs are fabricated in MOS technology. MOS stands for metal, oxide, semiconductor; an enumeration of the materials of the gate, the insulator, and the channel under the gate. The advantages of MOSFET over bipolar technology are manifold. Foremost, it allows for denser packing and is therefore the preferred technology for very

highly integrated circuits. Another advantage is its low power consumption. The gate is isolated from the other electrodes; the only current drawn at the gate is for charging and discharging the capacitance which the gate forms with the substrate. In contrast to the bipolar transistor, the FET has a very high input impedance.

On the other hand, the advantage of the bipolar technology is its lower output impedance, i.e. its higher driving force. It is therefore the preferred technology for components connected by (possibly long) wires on a circuit board. A combination of bipolar and MOS technologies is called BiCMOS, and it allows the use of FETs for chip-internal parts and bipolar transistors for components connecting to external components via pins. It is not hard to guess that the price for this feat is a more complicated fabrication process.

Inverters and Nand gates are built in practically the same fashion as with bipolar transistors. Figure 1.8 depicts symbolically an inverter, a Nand, and also a Nor gate. The output of the Nor gate is low, if any input is high, and high otherwise.

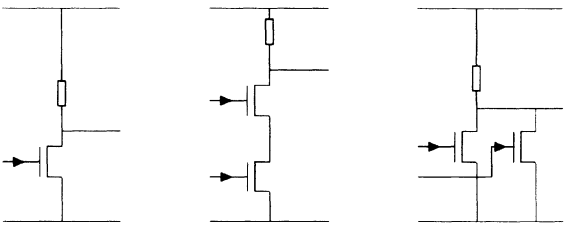


*Fig. 1.8. Inverter, Nand, and Nor gates using NMOS*

The technology based on npn (n-channel) transistors is called NMOS. We note that in case of a low output, i.e. when a transistors shuts, a current i = V/R flows, whereas if the output voltage is high, the transistor is open and no current is drawn. It is not only this asymmetry of current consumption that is unpleasant, but more so the fact that current flows and energy is consumed (i.e. converted to heat) even when a circuit is at rest.

*CMOS*

The prevailing technology now is CMOS. It eliminates the mentioned drawbacks by using n-channel and p-channel FETs in a fully symmetric arrangement. Its disadvantage is that twice as many transistors are required, and that a more sophisticated fabrication

process is necessary to accommodate both kinds of transistors on the same substrate. Nevertheless, its advantage is so pronounced that NMOS technology has almost disappeared. Figure 1.9 shows the gates of the previous figure implemented in CMOS technology.

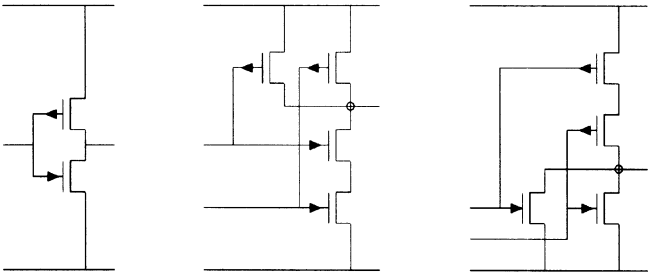


*Fig. 1.9. Inverter, Nand, and Nor gates using CMOS*

Considering for example the inverter, one might rejoice at having found the ideal, zero-energy circuit technology. After all, in both states, one of the two transistors is open, and since they are in series, no current will ever flow. The fallacy of this argument lies in considering static states only. A current flows as soon as a state transition occurs. It is due to the fact that every wire (lead, connection) represents a certain inherent, parasitic capacitance (see Fig. 1.10).

*Parasitic capacitance*

Assume the inverter to be in the state "output low". When the input switches from high to low, the lower transistor opens and the upper transistor shuts. The rising output voltage implies that the capacitance must be charged via the upper transistor by the amount Q = C*dV, where dV is the voltage difference between low and high states. This charge represents a current from the supply into the capacitance. During the next state transition from a high to a low output, the capacitance is discharged via the lower transistor.

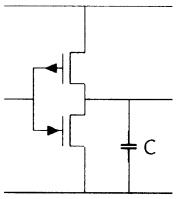



*Fig. 1.10. Inverter with inherent capacitance shown*

The argument shows that the current consumption of CMOS circuits is (approximately) proportional to the size of the inherent capacitance, to the difference in potentials representing 0 and 1, and also to the frequency of state transitions. The capacitance can be reduced by a reduction of the dimensions of the circuit components (miniaturization), and the potential difference (voltage swing) by reducing the operating (supply) voltage. The latter, together with a lower energy consumption, is the reason for the trend from the standard supply voltage of 5V to a new standard of 3.3V.

It follows that current is consumed only at the moment when a signal value changes. In many, if not most circuits, all changes occur at the same instant, resulting in short but large current peaks. These peaks may cause the supply voltage to ripple accordingly, with unforseeable consequences (feedback). In order to avoid this effect, it is recommended to place a decoupling capacitor between the supply and ground immediately adjacent to each integrated circuit (chip). This capacitor supplies the large instantaneous current needed to charge the output capacitance. The typical value of a decoupling capacitor is 0.1 μF.

*Decoupling capacitor*

It must also be pointed out that "ground", i.e. the zero potential common to all circuit components, is a convenient *abstraction,* but is only approximately realizable in practice. This is because "ground" between components is established by wires of non-zero length, implying non-zero resistance and inductance, giving rise to voltage differences across the connection. It is therefore mandatory to keep such ground connections as short as possible (low inductance) and to use wires of reasonable size (low resistance). If a circuit is mounted on a printed circuit board, it is highly recommended to use a specific ground layer, ensuring low resistance between all ground connections. A ground layer is indispensible if high frequency signals occur.

*Ground layer*

### 1.3. Electrical Characteristics of Gates

As noted before, every connection carries an inherent, parasitic capacitance which is to be charged or discharged upon every voltage change. Because transistors are not ideal switches, but have a finite,

although small resistance when shut, every gate forms an RC element as displayed in Fig. 1.11. Its effect is that the output voltage reaches the threshold a certain time span later than the input voltage passes its threshold. Gates therefore inherently cause a delay in their signal propagation.

*Propagation delay*

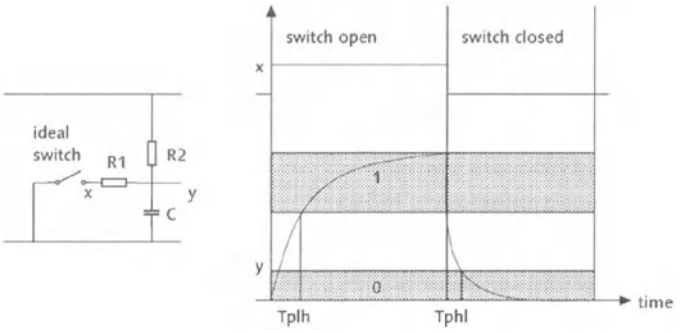



*Fig. 1.11*
*Model of gate and output voltage*

In general, the delays in the two directions of signal change are slightly different, since R1 and R2 are different. They are denoted by $T_{plh}$ (propagation delay from low to high) and $T_{phl}$ (propagation delay from high to low). Table 1.1 lists typical values for some widely used technologies, namely LS (low power Schottky), ALS (advanced low power Schottky), and CMOS.

Voltages for the low and high levels also differ among the various technologies. This is relevant in particular when circuits are designed using components of several technologies (mixed design). Figure 1.12 displays the levels for both inputs and outputs. We note that the two levels are wider apart for output than for input signals. This guarantees defect and noise immunity.

The diagram at the right in Fig. 1.12 shows an output from a bipolar component (TTL) connected to an input of a CMOS circuit. Here the output denoting "high" is lower than the input which is guaranteed to be recognized as "high". Evidently this combination must be avoided, and a so-called *level-shifter*, an adapter, must be inserted between a TTL output and a CMOS input.

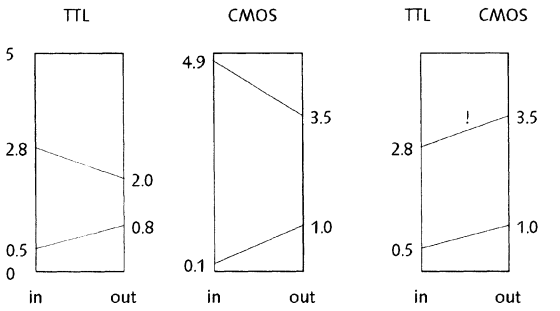



*Fig. 1.12 Voltage levels for TTL and CMOS circuits*

We here refer to components demanding a supply voltage of 5V (4.5 – 5.5V). More recently, a standard is advancing that requires a 3.3V supply only. Its advantage is a lower power consumption and smaller propagation delays. The lower voltage swing (voltage difference between the two levels) reduces the charge Q with which parasitic capacitances have to be charged or discharged, and thereby the current drawn upon state changes.

*Voltage swing*

Since every output transistor has a small, non-zero internal resistance, the output current that can be drawn is limited. And since every input represents a large, but finite resistance, it draws a small, non-zero current. As a consequence, only a limited number of inputs may be connected to an output, namely as many as the output transistor is able to "drive". Their maximum number is called the *fanout*. Typical fanout values are given in Tab. 1.1

*Table 1.1. Timing and voltage characteristics of TTL and CMOS technology devices*

| | TTL-LS | TTL-ALS | CMOS | |
|---|---|---|---|---|
| $T_{plh}$ $T_{phl}$ | 10 | 4 | 2 | ns |
| Vil | 0.8 | 0.8 | 1.0 | V |
| Vih | 2.0 | 2.0 | 3.5 | V |
| Vol | 0.8 | 0.4 | 0.05 | V |
| Voh | 2.5 | 2.5 | 4.95 | V |
| fanout | 20 | 20 | 50 | |

**Summary**

Bipolar transistors are the elements of TTL-technology. Electrons flow from the emitter to the collector under control of the base, and holes flow in the opposite direction. Bipolar transistors act as current amplifiers. In MOS transistors, electrons flow from the source to the drain under control of the electric field of the gate. The gate is electrically isolated from the source-drain path, and therefore the input impedance is very high. Effectively, the gate presents a capacitive load only.

TTL-technology is used when relatively high driving power is required, although power versions of field effect transistors (Power MOSFETs) gain in acceptance.

CMOS technology uses pairs of complementary npn and pnp transistors, thereby preventing current flow in all static states. Current flows only when states change. Hence CMOS is the preferred low-power technology.

In digital circuits, transistors operate at two points of their characteristic only: either they are turned off (no current flow) or saturated. In both cases, slight changes in the input have no effect on the output. This is the reason for the insensitivity of digital circuits against disturbances.

# Combinational Circuits

# 2.

**Overview**

Gates are circuit components representing the elementary Boolean operations of negation, conjunction (and), and disjunction (or). Combinational circuits are trees of gates, thus implementing general Boolean functions. Some reasonably simple functions - and therefore patterns of gates - occur very frequently in practical use. Among them are the decoder, the multiplexer, the adder and - to a lesser degree - the multiplier. But also the important read-only memory (ROM) falls into the class of combinational circuits.

## 2.1 Boolean Algebra

*Truth values*

Boolean Algebra is the calculus based on logical values, also called *truth values*, and logical operators. It was postulated by the mathematician George Boole (1815-1864). Since digital circuits operate with two possible values for each signal, Boolean Algebra is an adequate formal basis for digital circuits. Various notations have been established in different fields of application. One should be aware that they denote the same entities.

The logical calculus deals with two values only for each variable:

| Algebra | Programming | Circuit design |
|---|---|---|
| T | TRUE | H (high) 1 |
| F | FALSE | L (low) 0 |

Although there exist 16 possible binary operators combining two truth values x, y and yielding a truth value, only three are recognized as basic in practice. To them we add the only possible operation with one operand x (ignoring the identity operation):

| | Algebra | Programming | Circuit design | |
|---|---|---|---|---|
| Negation (not) | $\neg x$ | ~x | $\overline{x}$ | ~x |
| Conjunction (and) | $x \wedge y$ | x & y | xy | x*y |
| Disjunction (or) | $x \vee y$ | x OR y | x + y | x + y |
| Equivalence | $x \equiv y$ | x = y | | |
| Exclusive disjunction (xor) | | | $x \oplus y$ | x - y |

*Note: Exclusive disjunction is the inverse of equivalence; it is also called* exclusive or.

In this text we adopt the notation displayed in the last column. The values defined by the operators are given as follows:

| x | y | ~x | x*y | x + y | x - y |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |

Boolean Algebra postulates certain properties or laws (rules) that are useful when manipulating formulas. They are very similar to those of numerical algebra. We list some of these laws which are relevant for purposes of circuit design. They apply when equivalent formulas (expressions) are to be found which, for example, contain fewer operators, i.e. which simplify the expression and thereby the circuit which they represent.

| | | |
|---|---|---|
| ~(~x) | = x | |
| x + 0 | = x | |
| x + 1 | = 1 | |
| x * 0 | = 0 | |
| x * 1 | = x | |
| x * y | = y * x | commutative laws |
| x + y | = y + x | |
| x - y | = y - x | |
| (x * y) * z | = x * (y * z) | associative laws |
| (x + y) + z | = x + (y + z) | |
| (x - y) - z | = x - (y - z) | |
| (x + y) * z | = (x * z) + (y * z) | distributive laws |
| (x * y) + z | = (x + z) * (y + z) | |
| ~x * ~y | = ~(x + y) | de Morgan's laws |
| ~x + ~y | = ~(x * y) | |

*Combinational circuit*

Evidently digital circuits consisting of combinations of gates can be defined in terms of *Boolean expressions,* i.e. of expressions consisting of Boolean valued variables and of Boolean operators. A circuit is said to be *combinational* if it can be described by an expression. This implies that the circuit is free of loops, of feedback paths.

## 2.2. Graphical Notations

Very frequently, circuits are depicted graphically. The reason is that an expression's operators correspond to gates, and variables to signals represented by wires. The graphical description thus directly mirrors the laid-out circuit.

Also among graphical notations various standards have established themselves. We indicate the two more frequently used ones. In this text, we adopt the notation in the upper row of Fig. 2.1, as it is most widely applied. The symbols in the lower row are those of a more recently postulated IEEE standard.

*IEEE standard*

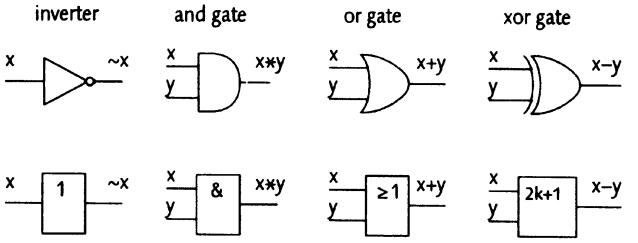



*Fig. 2.1. Graphical symbols for elementary operators*

When using graphics, inverters are usually combined with other gates and represented by small circles (like in the symbol for the inverter above). Typical are the inverted *and* and *or* gates (see Fig. 2.2), called *nand* and *nor* gates. As we know from the previous chapter, they are the truly elementary components from the point of view of component design. The *nand* operator is also called the *Sheffer stroke*, and *nor* is called the *Pierce operator*.

*Sheffer stroke*
*Pierce operator*

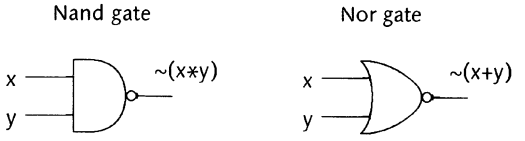



*Fig. 2.2. Graphical symbols for* nand *and* nor *gates*

## 2.3. Circuit Simplification

Circuit simplification is the task to find a circuit which is functionally equivalent to a given circuit, but simpler in some sense. The traditional criterion is the number of gates, and a significant number of methods and algorithms have been established for this purpose. With

modern technologies, however, it is often not the number of gates that counts most heavily, but for instance, the number or the length of wires. This radically changes the process of simplification. We therefore refrain from elaborating this subject in depth; it is well covered in many other textbooks. Instead, we list a few simple rules that are widely used and often sufficient for the task at hand.

*Introduction of auxiliary variables*

The first method to be mentioned is the introduction of auxiliary variables, or the sharing of signals. Consider, for example, the two definitions

$$a = x*y + z*w \qquad \text{and} \quad b = \sim(x*y) + \sim(z*w)$$

We introduce the auxiliary variables $u = x*y$ and $v = z*w$. Then $a = u+v$ and $b = \sim u + \sim v$ (Fig. 2.3).

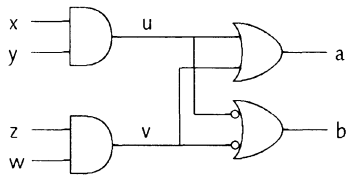



*Fig. 2.3. Simplified circuit for a and b*

*Duality principle*

Another, very important method rests on de Morgan's laws, which were listed above, and are here shown in terms of expressions and graphical symbols (Fig. 2.4). This law is also called the duality principle, expressing the fact that the and and or functions can be systematically exchanged if all signals are considered as inverted.

In practice, this implies that an equivalent (not necessarily simpler) circuit is obtained by systematically drawing inverters where there were none, and vice versa, and replacing *and* gates by *or* gates, and vice-versa. The example of Fig. 2.5 illustrates this cookbook method.

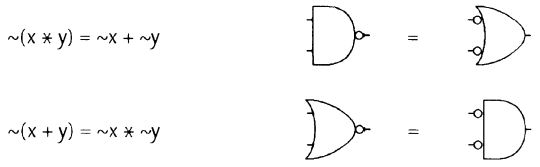



*Fig. 2.4. The duality of* and *and* or *gates*

Here we must recall that in terms of electronics, *nand* gates are somewhat simpler than *and* and *or* gates; hence the above transformation indeed leads to a simpler circuit. It appears as simpler, however,

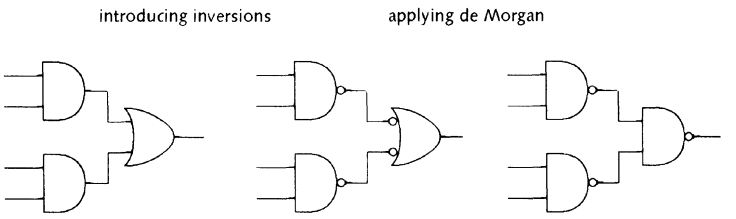



*Fig. 2.5. Applying de Morgan's law*

A second example shows how to treat the *exclusive or* gate. First we recognize that

$$x - y = x * \sim y + \sim x * y$$

and then simplify the circuit similarly as above (Fig. 2.6).

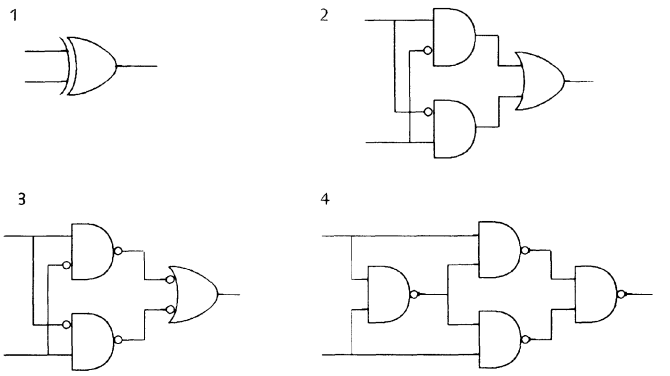



*Fig. 2.6. Representing the* xor *gate with an equivalent circuit consisting of 4* nand *gates*

*Elimination of redundant terms*

The third and last simplification method to be mentioned here is the elimination of redundant terms. It results from the application of the following equivalences:

$$\sim(\sim x) = x \qquad x + \sim x*y = x + y$$
$$x*y + \sim x*y = y \qquad x * (\sim x + y) = x*y$$

There are various methods (algorithms) in wide use for the simplification of Boolean expressions. Perhaps best known is the method proposed by *M. Karnaugh* in 1953. It consists of repeated application

of the equality

$$x*y + \sim x*y = y$$

and the use of a tabular representation of the expression in order to facilitate the recognition of cases where the equality is applicable. The expression is stated in the form of a map with $2^n$ cells, where n is the number of variables. Each cell specifies the value of the function for a unique combination of the argument values. The map is called a *Karnaugh-map* and effectively represents the truth table of the expression. Fig. 2.7. shows K-maps for 2, 3, and 4 variables (note we write xy for x*y).

*Karnaugh-map*

| | a | ~a |
|---|---|---|
| b | 1 | 1 |
| ~b | 1 | 0 |

f2 = ab + ~ab + a~b

| | ab | ~ab | ~a~b | a~b |
|---|---|---|---|---|
| c | 1 | 1 | 1 | 1 |
| ~c | 0 | 0 | 1 | 0 |

f3 = abc + ~abc + ~a~bc + ~a~b~c + a~bc

| | ab | ~ab | ~a~b | a~b |
|---|---|---|---|---|
| cd | 0 | 1 | 0 | 0 |
| ~cd | 1 | 1 | 1 | 1 |
| ~c~d | 0 | 1 | 0 | 0 |
| c~d | 0 | 1 | 0 | 0 |

f4 = ~abcd + ab~cd + ~ab~cd + ~a~b~cd
+ a~b~cd + ~ab~c~d + ~abc~d

*Fig. 2.7. Karnaugh maps for functions of 2, 3, and 4 variables*

The peculiarity of K-maps is that their rows and columns are labelled in such an order that cells which differ by a single argument value are adjacent (assuming that rows and columns "wrap around"). Hence, whenever two adjacent cells contain a 1, they can be "merged" by application of the mentioned equality, and thus a variable can be eliminated. The Karnaugh method is illustrated by Fig. 2.8 when applied to f3 of Fig. 2.7.

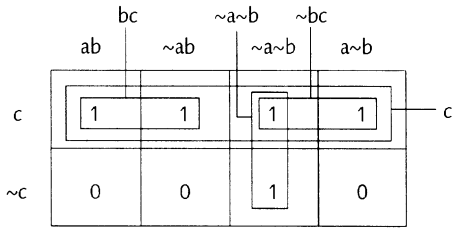



*Fig. 2.8. Combining adjacent 1-cells in the Karnaugh map of f3*

The result is obtained as a minimal covering (sum) of all cells containing a 1 (1-cells), and in this case it is

```
f3 = ~a*~b + c
```

Applying the same procedure to the function f4 of Fig. 2.7 results in

```
f4 = ~a*b + ~c*d
```

Obviously, the Karnaugh method has its severe limitations: expressions with more than 4 variables require a map represented in more than 2 dimensions. The advantage of the easy visibility of adjacent 1-cells is diminished, if not lost. In fact, K-maps are adequate tools for use "by hand" for simple cases. More complex cases are solved by formal methods, typically represented by algorithms performed by computers. The best known among these is the Quine-McCluskey method. We refrain from elaborating on it, because for the practitioner it is available in the form of simplification programs.

Quine-McCluskey

## 2.4. The Decoder or Demultiplexer

The building blocks of digital circuits are typically larger units than individual gates. Certain combinations of a relatively small number of gates represent functions that occur very frequently and constitute what might be called a second level of elementary circuits. In this and the following subchapters, we present the most frequently encountered of these units.

Demultiplexing is the operation of distributing a source signal x onto several destinations $y_i$ according to the value of a *selector signal* s. Evidently, s denotes an index, a number. In order to derive a circuit for a demultiplexer, it is necessary to postulate an encoding of inte-

*Selector signal*

*Binary encoding*

gers in terms of digital signals. The standard encoding is the so-called *binary encoding*. It rests on the assumption that the digital value of a signal ("0" or " 1") is taken as a numeric value, as a *bi*nary digi*t* (bit), and that each signal component ($s_0$, $s_1$, ...) is a weighted term in the sum s, namely

$$s = s_0*2^0 + s_1*2^1 + s_2*2^2 + \ ... \ + s_i*2^i + \ ...$$

Then the demultiplexer function is expressed as

$$y_i = (\text{if } i = s \text{ then } x \text{ else } 0) \quad y_i = x * (i{=}s)$$

The resulting circuit with 2 outputs is shown in Fig. 2.9 together with the symbol used for the demultiplexer.

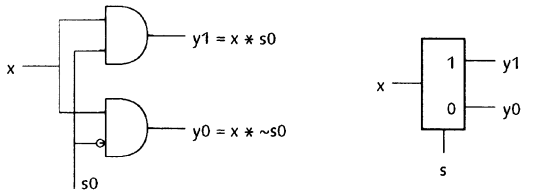



*Fig. 2.9. Demultiplexer circuit and symbol*

*Cascading*

In order to obtain a demultiplexer with a larger number of outputs, we employ the important technique of cascading. Indeed, a demultiplexer with $2^n$ outputs is easily obtained by cascading n levels of demultiplexers with 2 outputs. This is shown in Fig. 2.10. It is common to speak of an n-to-$2^n$ demultiplexer.

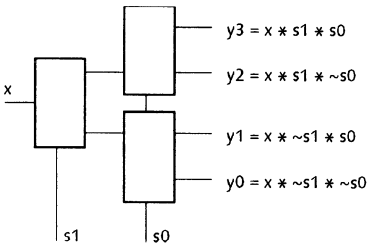



*Fig. 2.10. Cascaded demultiplexer*

If the input x is held constant with value 1, then $y_s = 1$ with all other outputs being 0. In this case, the demultiplexer acts as a *decoder* of s (and x is called an *enable signal*). The terms demultiplexer and decoder are, however, frequently used as synonyms in practice.

## 2.5. The Multiplexer

A second standard building element is the *multiplexer,* the inverse of the demultiplexer. Its function is to unite several sources xi into a single destination y according to a selector signal s. The multiplexer is therefore also called a selector. The multiplexing function is defined as $y = x_i$.

*Multiplexer also called a selector*

We again start by considering the simple case with 2 sources, where $s = s_0$. $y = x_i$ is transformed into

$$y = MUX(s, x_0, x_1) = (\text{if s then } x_1 \text{ else } x_0) = x_0 * \sim s + x_1 * s$$

The 2-input multiplexer circuit and its symbol are shown in Fig. 2.11.

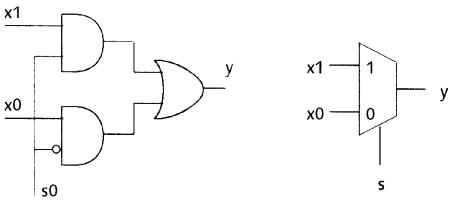



*Fig. 2.11. Circuit and symbol for 2-to-1 multiplexer*

By cascading n levels of 2-input multiplexers, a multiplexer with $2^n$ inputs is obtained (see Fig. 2.12). This process corresponds to functional composition.

$$MUX(s_1, MUX(s_0, x_0, x_1), MUX(s_0, x_2, x_3))$$

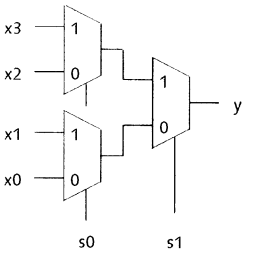



*Fig. 2.12. Cascaded multiplexers*

## 2.6. The Adder

The next standard building block to be introduced is the adder based on binary encoding of integers. The invaluable advantage of the binary encoding is that all digits can be treated in the same way, i.e.

by the same circuit. We may therefore start by considering the simplest case, namely that of adding two one-digit numbers x and y. The maximum result is 2, obviously requiring two bits (signals) for its representation.

| x | y | sum | c | s |
|---|---|---|---|---|
| 0 | 0 | 00 | 0 | 0 |
| 0 | 1 | 01 | 0 | 1 |
| 1 | 0 | 01 | 0 | 1 |
| 1 | 1 | 10 | 1 | 0 |

*Half-adder*

We call the two resulting signals s (sum) and c (carry), and recognize that they correspond to the two elementary functions of the *exclusive or* and the *and* respectively. This circuit is called a *half-adder* (Fig. 2.13). The reason for this name will become apparent when full adders are presented below.

$$s = x \cdot y \qquad c = x * y$$

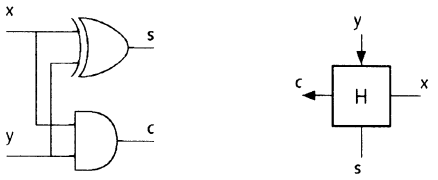



*Fig. 2.13. Half-adder element*

*Incrementer*

Cascading half-adders leads to an *incrementer*. We recall that in a positional number system, the input carry of a position is the output carry of the next lower position. Hence, the carry output is connected to one of the inputs of the half-adder in the next higher position (see Fig. 2.14). The incrementer adds the carry input of the lowest position to the inputs $x_0 \ldots x_{n-1}$, i.e. is capable of adding 0 or 1.

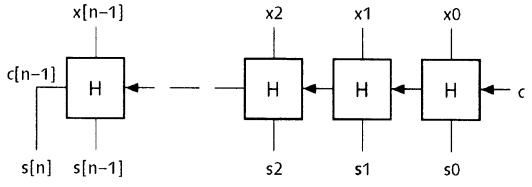



*Fig. 2.14. Incrementer*

A *full adder* must evidently consist of elements capable of adding three binary inputs, namely the two operands x and y, and the carry from the next lower element. A full adder element may obviously be built by joining two half adders as shown in Fig. 2.15. The function of the output carry is obtained by considering the eight possible input combinations with the aid of auxiliary variables u = x-y, v = x*y, and $w = u*c_{in}$.

*Full adder*

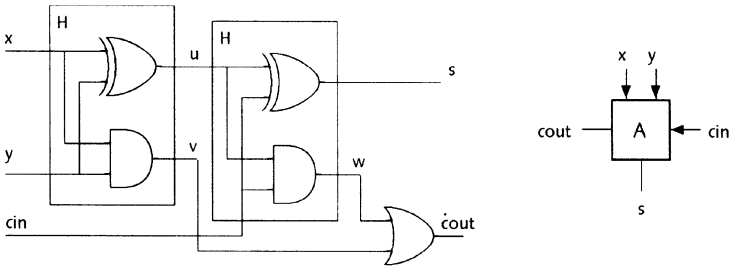



*Fig. 2.15 Full adder element*

| x | y | cin | u | v | w | cout | s | sum |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 2 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 2 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 3 |

An adder for two n-bit addends is obtained by cascading n full-adder elements and connecting the carry output to the carry input of the next higher stage and with $c_{in} = 0$, as shown in Fig. 2.16. In passing we note that the carry of the last position passes n elements and thereby 3n gates. For larger n, this path length easily becomes critical in circuits in the sense that the delay caused by the carry is a critical factor in the speed of an entire circuit.

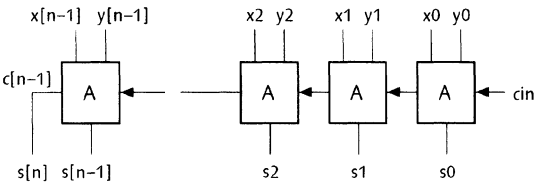



*Fig. 2.16. N-bit adder*

The design of a subtractor depends on the chosen representation of negative numbers. Desirable is a form where the weights of the digits are independent of a number's sign, because then it is possible to use exactly the same circuit as both adder and subtractor, relying on the equality x - y = x + (-y). This condition is satisfied by the 2's-complement notation, which therefore has not only become popular, but is used practically without exception. It attributes a negative weight to the highest digit $x_{n-1}$. In order that a number x be negative, $x_{n-1}$ must be 1. If $x_{n-1}$ is 0, all terms are positive, and therefore also x is positive. Evidently, $x_{n-1}$ alone determines the sign of a number; it is therefore called the sign bit.

*2's-complement notation*

$$x = -x_{n-1}\,2^{n-1} + x_{n-2}\,2^{n-2} + \ldots + x_1\,2^1 + x_0\,2^0$$

Sign inversion is achieved by logically inverting all digits, i.e. by replacing all $x_i$ by $1-x_i$, and by adding 1 (using $c_{in}$ for this purpose):

$$\begin{aligned}
&-(1-x_{n-1})\,2^{n-1} + (1-x_{n-2})\,2^{n-2} + \ldots + (1-x_1)\,2^1 + (1-x_0)\,2^0 + 1 = \\
&-(1-x_{n-1})\,2^{n-1} + 2^{n-1} - (x_{n-2}\,2^{n-2} + \ldots + x_1\,2^1 + x_0\,2^0) = \\
&-(-x_{n-1}\,2^{n-1} + x_{n-2}\,2^{n-2} + \ldots + x_1\,2^1 + x_0\,2^0) = -x
\end{aligned}$$

*Overflow*

If a sum (or difference) cannot be represented by n digits, an overflow is said to occur. Whereas in adding two unsigned numbers a carry output with value 1 signals overflow, the carry cannot be interpreted in this sense in the case of signed numbers. Rather, overflow is indicated by the carry outputs of the highest (sign) and the second highest digit being different. Evidently, a single xor gate suffices for this purpose.

unsigned arithmetic: $ov = c_{n-1}$
signed arithmetic: $ov = c_{n-1} - c_{n-2}$

## 2.7. The Adder with Fast Carry Generation

As previously mentioned, the presented adder has one rather problematic property: the carry output is a signal rippling through all adder elements starting with the lowest digit. This implies that the signal delay depends linearly on the number n of digits involved, and that the circuit is slow for large n. We know that in principle every Boolean function can be expressed in normal form, which is implementable by a layer of (multi-input) *and* gates followed by a layer of (multi-input) *or* gates, hence resulting in a minimal path length of 2 independent of the function's complexity. As it turns out, however, the number of gates grows exponentially with n, making this solution prohibitive in practice.

A realistic solution which reduces the path length substantially while at the same time requiring only a moderate increase of gate resources is based on the following observation and has become known as *fast carry generation*:

*Fast carry generation*

At each stage in the chain of adder elements a carry may either be generated or propagated. A carry is generated if both addends have value 1; we define $g = x * y$. A carry is propagated if any of the addends has value 1; we define $p = x + y$. At every stage, the carry is thus defined by

$$c_i = g_i + p_i c_{i-1} \text{ with } g_i = x_i * y_i \text{ and } p_i = x_i + y_i$$

Let us now build an adder for 4-bit numbers using this idea and restricting our attention to carry generation. We obtain:

$$c_0 = g_0 + p_0 c_{in}$$
$$c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{in}$$
$$c_2 = g_2 + p_2 c_1 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$$
$$c_3 = g_3 + p_3 c_2 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$$

The resulting new adder, consisting of 4 elements, each generating a sum, a generate and a propagate signal, and of a fast carry generator is shown schematically in Fig. 2.17.

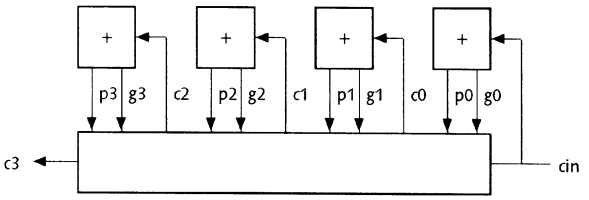



*Fig. 2.17. Fast carry generation unit*

Like in the case of the adder element, we now let also the fast carry generator itself produce a propagate and a generate signal instead of a carry output, such that

$$c_3 = g + p\, c_{in}$$

p and g are obtained from the expression for $c_3$ as

$$g = g_3 + p_3 c_2 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$
$$p = p_3 p_2 p_1 p_0$$

With this alteration, the unit is now capable of being cascaded. For example, a fast adder for 16-bit numbers is built by using two levels of fast carry generators (Fig. 2.18), and one for 64-bit numbers using three levels. Evidently, the carry path length increases with the logarithm of n only. This is quite acceptable even for the most stringent requirements. Typically, 4-bit adders are available as units incorporating the first level of fast carry generation. A 16-bit adder then consists of 4 such units and a single fast carry generator.

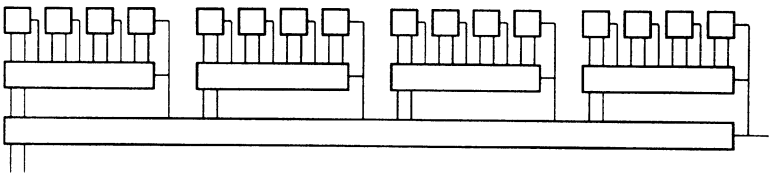


*Fig. 2. 18. Cascaded fast carry generators*

## 2.8. The Multiplier

Multiplication is inherently more complex than addition. Hence also the circuit implementing it should be expected to be significantly more elaborate. The use of a positional notation for numbers reduces multiplication to n additions, when n is the number of digits in the multiplier.

$$x = x_{n-1}*2^{n-1} + ... + x_2*2^n + x_1*2^1 + x_0*2^0$$
$$x*y = x_{n-1}*y*2^{n-1} + ... + x_2*y*2^n + x_1*y*2^1 + x_0*y*2^0$$

We have thereby reduced multiplication of two n-bit operands to n-1 additions of n-bit operands and n multiplications of an n-bit multiplicand y by a single-bit multiplier $x_i$. The latter turns out to be identical to *and* operations. The concept is schematically illustrated in Fig. 2.19, displaying the weights (powers of 2) as appropriate shifts of the multiplicand.

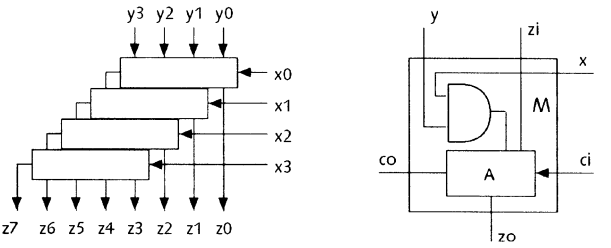



*Fig. 2.19. Multiplier schematic and multiplier element*

Considering the multiplier as a matrix M of elements with index i running from top to bottom and index j running from right to left, the connections are specified by the following equations:

M[i, j].x = x[i]
M[i, j].y = y[j]
M[i, j].zi = M[i-1, j+1].zo  M[0, j].zi = 0
M[i, j].ci = M[i, j-1].co  M[i, 0].ci = 0
M[i, N-1].zi = M[i-1, N-1].co

As evident from Fig. 2.20, full multipliers require a large amount of circuitry. Therefore, they are typically implemented as a sequence of additions executed sequentially in time. The exception are computers

specifically designed for numerical computations where multiplication is a frequent operation, and where therefore the heavy expenses can be justified. The circuit becomes even more complex when measures for speeding up carry propagation are employed. We note that in the circuit of Fig. 2.20 the longest carry path contains 2n elements, i.e. 6n gates. The use of fast carry propagation therefore appears to be advisable.

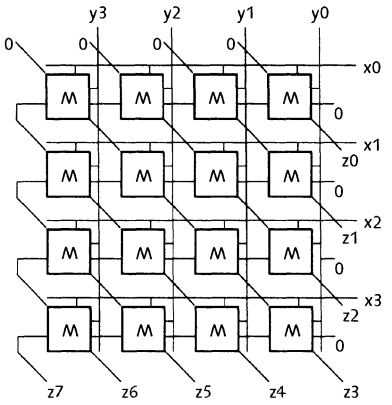



*Fig. 2.20. 4 x 4 bit multiplier*

## 2.9. The Read-Only Memory (ROM)

Another frequently encountered purely combinational circuit is the read-only memory. Its structure is such that any Boolean function of n variables, where n is the number of its inputs, can be generated. Since complicated functions are usually not perceived as functions, but rather as individual values corresponding to the possible (combinations of) input values, the device is called a memory rather than a function generator.

A ROM essentially consists of a decoder of the binary-encoded input number, called the *address,* an array of *or* gates, and a set of output drivers. The decoder yields a selector signal for each input value, "addressing" each "cell" as shown in Fig. 2.21. If there are n input signals, there are $2^n$ selector signals. Typical ROM chips have m = 8 outputs, i.e. generate m functions simultaneously using the same decoder.

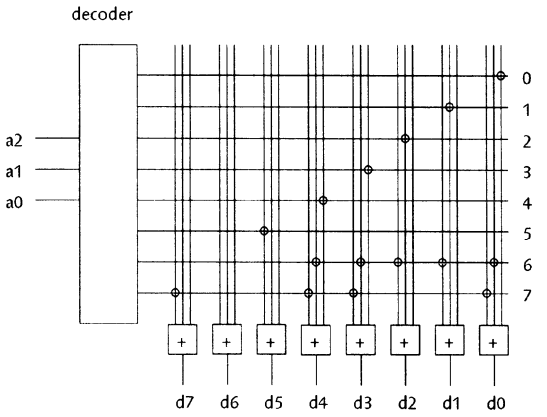



*Fig. 2.21. Structure of a ROM with $n = 3$ and $m = 8$*

Connections are exhibited in Fig. 2.21 by small circles, and vertical lines without any connections are assumed to carry the value 1. The connections shown represent the following (hexadecimal) values, where d[a] stands for the output d with input a:

d[0] = 01, d[1] = 02, d[2] = 04, d[3] = 08,
d[4] = 10, d[5] = 20, d[6] = 1F, d[7] = 99

Typically, such structures are drawn using an abbreviated notation as shown in Fig. 2.22: the inputs to the *or* gates are collected into a single line. The meaning of Fig. 2.22, however, is the same as that of Fig. 2.21.

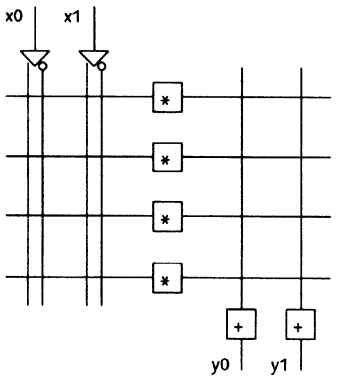



*Fig. 2. 22. Structure of a ROM using abbreviated notation*

The regular structure of the ROM circuit permits to achieve very high

densities of gates. Currently (1993) ROM chips are available with up to 4 million *or* connections, representing 4 MBits. This requires (with m = 8) 19 address inputs, and consequently a very large decoder. We recall that also decoders are implementable with highly regular structures of gates.

The fact that ROMs are regarded as memories instead of as function generators is supported by the various techniques offered for defining the function or, rather, storing the data. We note that all possible functions are achievable by changing the connections to the *or* gates only.

In the case of the (original) ROM, these connections are made when fabricating the chip. Nevertheless, the first fabrication steps produce a regular device with all possibilities left open. Only a later step determines the *or* connections, which can be specified by the individual customer. But the production of the so-called mask defining these connections is quite expensive, and is justified only when reasonably large quantities of the same chip are to be fabricated.

*Programmable ROM*

A much more practical device is the *programmable* ROM, called *PROM.* This device is delivered to customers with all possibilities still open. The required data are then stored by the customer "in the field"; this process is called *programming* the ROM. In the original device, all connections are closed; programming consists of breaking unwanted connections. It is achieved by applying a voltage and thereby a current such that the selected connection is burnt (like blowing a fuse). Devices used for programming are therefore often called PROM burners.

*Programming the ROM*

Once a PROM is programmed, it cannot be reprogrammed (except by blowing additional fuses). A more flexible solution is available in the form of the *erasable* PROM, called *EPROM.* In addition to being programmed, it can be erased, i.e. returned to its original state with *all* connections reestablished. This effect is achieved by using transistors with floating (isolated) gates, which retain their charges and potential for a very long time. During programming, the gates are charged by applying a higher than normal voltage, and discharged by ultraviolet radiation for several minutes.

*Erasable PROM*

*Floating gates*

An even more advanced technology makes erasure possible through electrical effects. Such an *electrically erasable* PROM is called *EEPROM.* In principle, an EEPROM, although still a combinational

*Electrically erasable PROM*

circuit, appears like a memory (RAM), with the exception that the time needed for storing data is several thousand times greater than the time required for reading data. (EEPROMs are also called Flash-ROMs.)

*Flash-ROMs*

## 2.10. The Combinational PLD

The fact that any function of a number of input variables can be implemented by the ROM structure is an alternate formulation of the rule that every Boolean function can be expressed as a sum of products, or alternatively, as a product of sums of the variables or their negations. If an expression is in one of these forms, it is said to be in normal form. A sum of products is called *disjunctive normal form,* and a product of sums is called *conjunctive normal form.* This truth is the basis of *programmable logic devices* (PLDs), to be discussed in this section. Subsequently, we will consider the disjunctive normal form only and simply call it *normal form.*

*Disjunctive normal form*
*Programmable logic device*

A general, programmable device, represents the entire family of functions of its inputs and directly reflects the normal form. Any individual function is selected by closing (or opening) connections of its *and* and *or* gates, i.e. by programming it in the sense of a ROM. The structure is shown in Fig. 2.23 for a device with 2 inputs and 2 outputs.

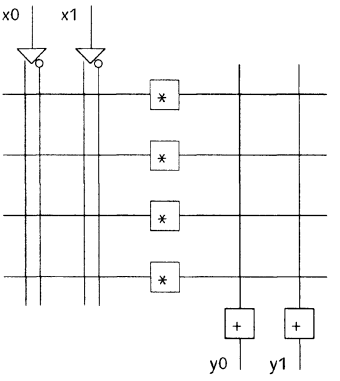



*Fig. 2.23. General structure of combinational PLD*

The left part, which constitutes a matrix containing *and* gates only, is called the *and matrix,* whereas the right part containing *or* gates only is called the *or matrix.* At this point we recognize the ROM to be a

special case of this structure, namely one where the *and* matrix is fixed as a complete decoder with $2^n$ outputs, whereas the *or* matrix is programmable and determines the data stored. The general structure with both matrices being programmable is called a *programmable logic array* (PLA), but is indeed rarely encountered in practice, because it is too general for most applications. However, the "opposite" of the ROM is gaining high importance as a flexible device, namely the above structure with the *and* matrix programmable and the *or* matrix fixed (all connections closed). It is known under the terms PAL or (combinational) PLD.

*Programmable logic array (PLA)*

*PAL, PLD*

Figure 2.24 shows the connections needed to implement various basic functions of two variables. Two cases are displayed, one for devices with positive outputs, the other for devices with inverted outputs.

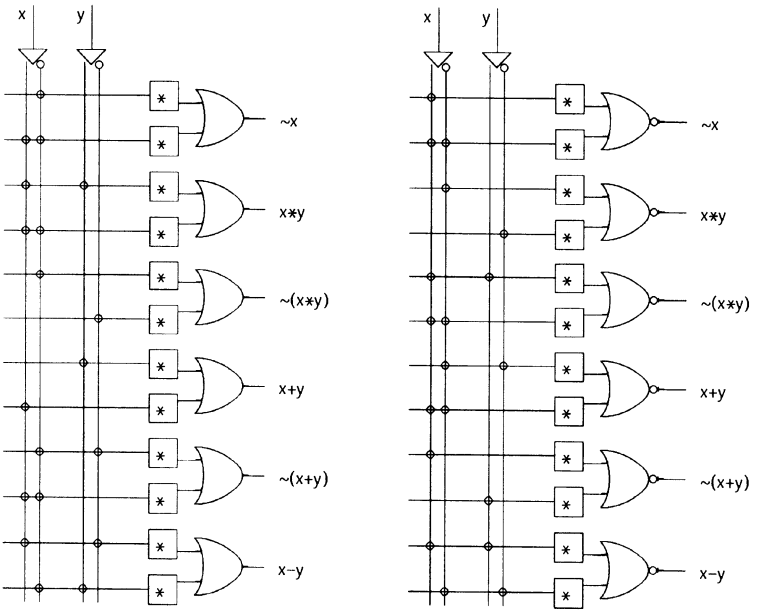



*Fig. 2.24. Basic functions implemented with PLDs*

Commercially available PLDs typically allow one to use outputs as additional factors in the product lines, thereby allowing the realization of functions that might otherwise be impossible. Figure 2.25

shows this arrangement. For example, the function displayed through connections in the *and* matrix could not be implemented with only 2 product lines (inputs to the *or* gate). The typical number of product terms, however, is 8.

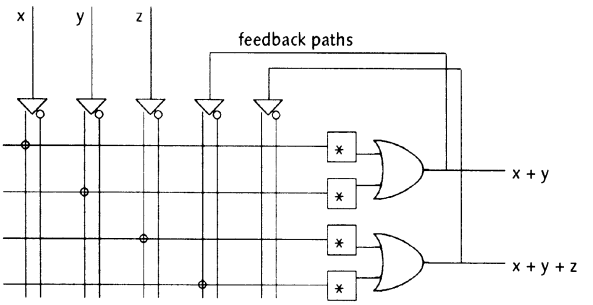



*Fig. 2.25. PLD with feedback paths*

As an additional example for the realization of a function with a PLD, Fig. 2.26 shows a full adder. The reader should compare it with Fig. 2.15.

$$s = x*\sim y*\sim c_{in} + \sim x*y*\sim c_{in} + \sim x*\sim y*c_{in} + x*y*c_{in}$$
$$c_{out} = x*y + x*c_{in} + y*_{cin}$$

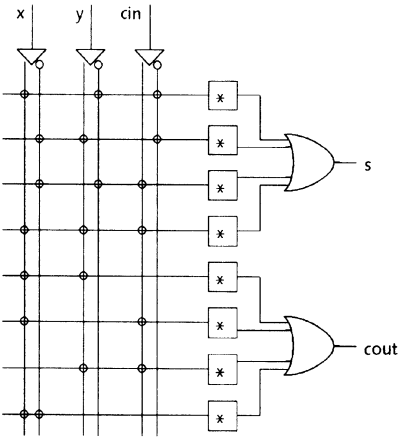



*Fig. 2.26. Adder represented by PLD*

Commercially available PLDs are defined (loaded, *programmed*) with the aid of computers. The respective software expects input in the form of an equation (in disjunctive normal form) for each output.

The device connected to the computer is called a *burner*, as it burns fuses in the case of one-time programmable components. For development purposes, components are available whose "burnt" connections can be reestablished and reprogrammed many times (EPLD). The erasure occurs either by applying ultraviolet light or electrically.

*EPLD*

## 2.11. The Programmable Gate Array

Another regular structure that is configurable by programming to user-specific demands is the *gate array*. It consists of an array (matrix) of identical cells. Here, both the function as well as the connections of each cell can be selected individually. These selections are held by memory elements associated with each cell. Two kinds of field programmable gate arrays (FPGA) are widely used: In the first kind, the memory elements are loadable and erasable as a whole, like in the EPROM. In the second kind, the memory cells form a static RAM (see Chap. 6), which implies that the *configuration* (functions and connections of cells) can be changed very rapidly and arbitrarily many times. This is particularly attractive for the development and testing of circuits.

*Field programmable gate arrays (FPGA)*

Naturally, in most applications a certain number of cells remains unused, and a particular function may have to be composed of several cells because the one desired is not directly available as a cell function. This may lead to a relatively poor utilization of the available resources. However, considering the fact that gates (transistors) have become extremely cheap, whereas development effort and time are costly, the use of FPGAs represents a growing trend and a preferred solution to reduce the number of discreet parts in complex circuitry. Consequently, parts containing individual gates are being used more and more rarely.

In order to understand the principal idea of the FPGA, let us consider as an example a cell which allows one to select (to program) any of the 16 possible functions of two variables x and y. The cell is shown in Fig. 2.27. The central element is a multiplexer whose inputs are selectable, again by multiplexers. The latter, however, are controlled by values stored in the underlying memory, i.e. they are fixed (until the FPGA is reconfigured). Their selection signals are therefore not shown.

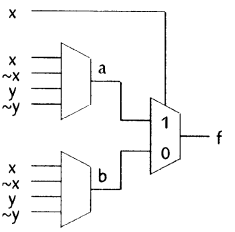



*Fig. 2.27. Example of a combinational FPGA cell*

The input selections for the 16 possible cases of functions of (at most) two arguments are the following (we realize that a cell with three inputs would be more realistic, as it would also offer the frequently used multiplexer function as a basic cell option):

| a | b | f | a | b | f | a | b | f | a | b | f |
|---|---|---|---|---|---|---|---|---|---|---|---|
| x | y | x+y | y | x | x*y | ~y | y | x-y | x | x | x |
| x | ~y | x+~y | ~y | x | x*~y | y | ~y | x-~y | ~x | ~x | ~x |
| y | ~x | ~x+y | ~x | y | ~x*y | x | ~x | 1 | y | y | y |
| ~y | ~x | ~x+~y | ~x | ~y | ~x*~y | ~x | x | 0 | ~y | ~y | ~y |

Designing a circuit with an FPGA involves not only the implementation of the desired functionality in terms of the cells, i.e. in selecting the cell functions, but also in connecting the cell inputs with the output of other cells. The latter task is called *routing*. It depends strongly on the available routing facilities, which vary considerably among available devices. Typically, every cell's output is available as input in at least the four neighbouring cells. Such an arrangement is shown in Fig. 2.28, with the cell of Fig. 2.27 augmented by four routing multiplexers.

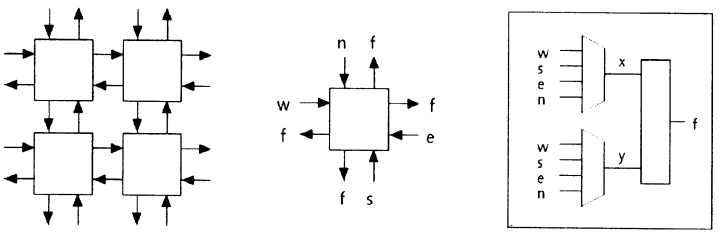



*Fig. 2.28. FPGA cells shown with connections to neighbouring cells*

## 2.12. Dynamic Behaviour of Combinational Circuits

As we have seen in the previous chapter, every gate causes a certain delay in the propagation of signal changes. In circuits with gates connected in series, these delays are additive and may become significant. Designers should therefore always aim at short signal paths. More will be said about this topic later.

Yet there is another problem that deserves to be, if not treated, then at least mentioned. To illustrate the problem, we consider the circuit as shown in Fig. 2.29.

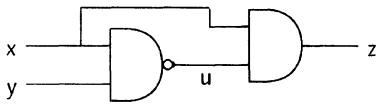



*Fig. 2.29. Circuit with hazard*

Furthermore, we consider the input signals x and y as functions of time, and we derive the gate outputs explicitly taking into account the gate delays for the outputs u and z (see Fig. 2.30).

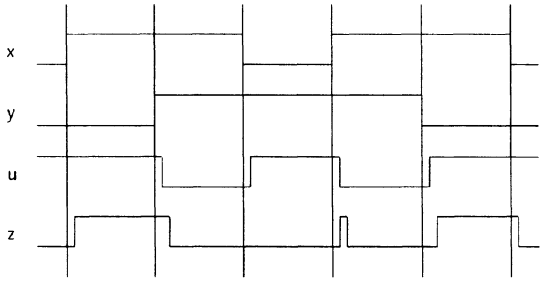



*Fig. 2.30. Occurrence of a hazard in z*

Not without surprise we note a short signal change of z after the fourth time period. It is caused by the *and* gate's one input to change from 0 to 1, and the other to change from 1 to 0, but with a slight time delay. Such a short, unintended signal is called a *hazard*, a *spike*, or a *glitch*. A hazard occurs at the output of a gate G, if the following conditions exist:

*Hazard, spike, glitch*

Let x0 and y0 be the values of G's inputs prior to their (synchronous) change, x1 and y1 the values thereafter. Let $t_x$ and $t_y$ be the propagation delays of the signals from their origin to the inputs of G. Then a hazard occurs if

1. $G(x0, y0) = G(x1, y1)$
2. $G(x0, y1) \neq G(x1, y0)$
3. $t_x \neq t_y$

There exist methods for avoiding glitches; they invariably result in more complex circuits. Here it must suffice to state that glitches can be ignored, if one is only interested in a signal's value at a time long enough after the last input signal's change. If this time period is chosen to be at least the delay caused by the longest signal path, then it is guaranteed that all outputs have reached a stable state, and hence glitches can safely be ignored. We will return to this rule in the discussion of sequential circuits in Chap. 4.

**Summary**

A combinational circuit consists of gates representing Boolean connectives; it is free of feedback loops. A combinational circuit has no state; its output depends solely on the momentary input values. In reality, however, signal changes propagate through a sequence of gates with a finite speed. This is due to the capacitive loads of the amplifying transistors. Hence circuits have a certain propagation delay.

Every Boolean function can be expressed in a normal form consisting of disjunctions (or-terms) consisting of conjunctions (and-factors), and it can therefore be implemented by two levels of gates only. Of considerable technical relevance are devices which represent two levels of gates in a general form. A specific function is selected by opening (or closing) connections between specific gates. This is called programming the device, and the device is a programmable device (PLD). Programming happens electrically under computer control. PLDs are highly attractive in order to reduce the number of discrete components in circuits. A specific form of a PLD is the read-only memory (ROM).

# Latches and Registers

3.

**Overview**

Combinational circuits lack the ability to store information. For this purpose, latches and registers are introduced. At a first glance, they are simple circuits consisting of a few gates only. Their distinguishing feature is a feedback loop. It makes the analysis of the circuit much more subtle and critical, and gate propagation delays play a crucial role. Our advice is therefore to restrict circuit design to combinational circuits, and to acquire latches and registers as complete parts. Feedback loops then exist only within these parts.

## 3.1. The SR-Latch

Combinational circuits are free of loops. We will now investigate the effect of allowing loops. To introduce loops in general, however, would dramatically complicate the problems of analysis, in particular timing, without yielding substantial benefits. We therefore concentrate our attention on a single, reasonably simple case which plays an important role and introduces a fundamentally new property, namely *retaining a state.* The circuit to be investigated is the *register.*

*State*

Let us first consider two inverters in series. The function represented by this circuit, i.e. its output voltage in terms of the applied input voltage is easily derived from the characteristic of a single inverter (see Fig. 3.1).

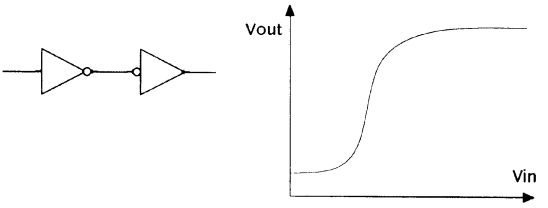



*Fig. 3.1. Characteristic of 2 inverters in series*

If we now connect the output of the second to the input of the first inverter, we apply the additional constraint $V_{in}=V_{out}$, represented by a straight line. It follows that the circuit can only be in one of three states, namely those where the straight line and the

characteristic intersect (Fig. 3.2).

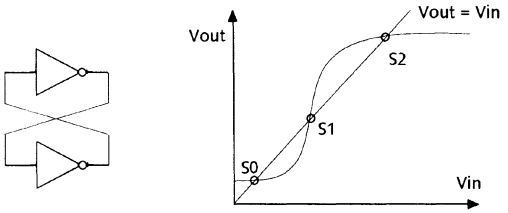



*Fig. 3.2. Inverters with feedback loop*

Closer inspection reveals that states S0 and S2 are both stable states. But if the circuit is in state S1, the slightest change in an input voltage, e.g. caused by induction of a surrounding field, by supply voltage change, or temperature change, is instantaneously amplified and transfers the circuit into either state S0 or S2. State S1 is therefore called metastable, which really means unstable. We recognize that the circuit has the capability of being stable in two states and therefore of holding *one bit of data.*

The circuit, however, is not really practical; after all, it has no inputs. This defect is easily remedied by replacing the inverters with *nand* gates as shown in Fig. 3.3. If both inputs have the value 1 (high), the new circuit corresponds to that of Fig. 3.2. If input S' is pulled low, output Q becomes 1 (is *set*), and if R' is pulled low, Q becomes 0 (is *reset*). The circuit is therefore called a *latch,* or more specifically an *SR-latch,* because it latches (holds) a state (bit).

*SR-latch*

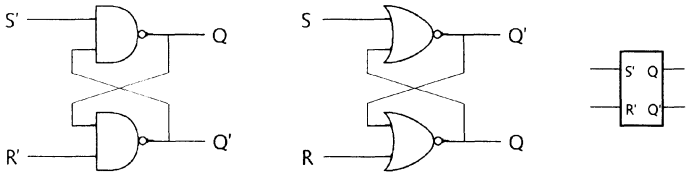



*Fig. 3.3. SR latches with* nand *or* nor *gates*

According to the duality principle, the operators *and* and *or* can be interchanged together with the polarity of all signals. This leads to the form of latch shown on the right in Fig. 3.3. In this case, the inputs are normally at 0; the set and the reset signals are therefore positive, also called *active high.* This is in contrast to the latch consisting of *nand* gates, where the input signals are said to be *active low.* We denote such

signals with an apostrophe, implying that the name stands for the signal's inverse. (Note that the apostrophe does not denote an operator, but is part of the name).

The dependent variable Q is given, respectively for the two versions shown in Fig. 3.3, by the formulas

$$Q = \sim(S' * \sim(R'*Q)) = \sim S' + R'*Q$$
$$Q = \sim(R + \sim(S+Q)) = \sim R * (S+Q)$$

Unless both inputs of the latch are active simultaneously - a case that should be considered as abnormal - Q' = ~Q holds.

## 3.2. The D-Latch

*D-latch*

The circuit that is much more typically used to hold a data bit is the *D-latch.* It is derived from the SR-latch by prefixing it with two additional gates as shown in Fig. 3.4. If the input G is held low, the circuit represents an SR-latch holding its state, since both S' and R' are high. If G is high, the output Q assumes the same value as input D. The latch is then called *transparent.* As a result, the latch output assumes the value of the D input as soon as G is made high, and it retains the acquired state as long as G is low. G is called the *enable* input, and D is considered as *data* input. The value of Q depends on G and D and is given by the truth table

| G | D | Q | |
|---|---|---|---|
| 0 | x | Q | storing |
| 1 | 0 | 0 | transparent, Q = D |
| 1 | 1 | 1 | |

and may be expressed by the formula

$$Q = G*D + \sim G*Q$$

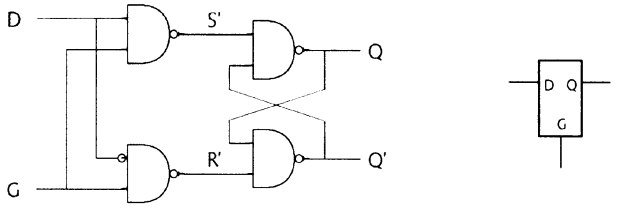



*Fig. 3.4. D-latch*

Closer scrutiny of the circuit of Fig. 3.4 shows that the fine details of its correct functioning depend on gate delays. As an exercise, let us try to design a latch by considering it as a multiplexer whose zero-input is its output (Fig. 3.5). We convince ourselves of its correctness as follows. If G is low, G' is high, the feedback loop BQ is closed and latches the data bit. From G being high, it follows that Q = ~A = D, i.e. the circuit is transparent. We notice that this is the circuit of Fig. 3.4. with the gate at the lower left replaced by an inverter, thus representing a slight simplification.

Statically the circuit is indeed a correct latch. But its dynamic behaviour is questionable as becomes apparent from the following argument: With its D-input constantly high, we consider a transition of G from high to low (see Fig. 3.5).

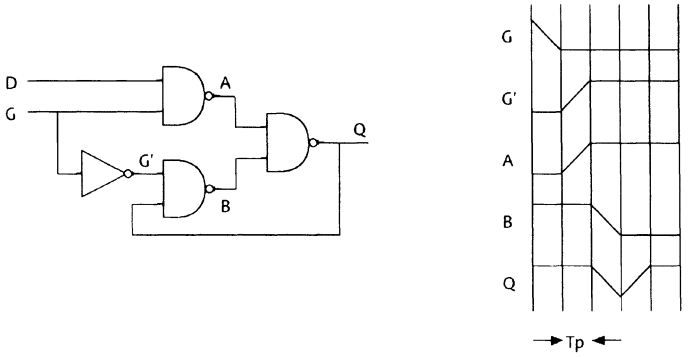



*Fig. 3.5. Design of an unreliable latch*

With some dismay we realize that, although the input is held constant, the latch output contains a hazard. Our analysis rests on the (generally unjustified) assumption that the gate delays of all gates are equal. If the inverter in Fig. 3.5 is considerably slower than the other gates, it may have the disastrous result that the output remains permanently low after the spike.

Hazards are typically eliminated by introducing logically redundant terms. In this case the single term D*Q leads to the hazard-free *Earle latch:*

*Earle latch*

$$Q = G*D + \sim G*Q + D*Q$$

The lesson to be remembered is the following: Never design a latch (or a register) yourself! Latches and registers are implemented using standard parts.

## 3.3. The D-Register

*transparency undesirable*

The transparency of a latch is often quite undesirable, particularly, if storage elements are to be arranged in series. Desirable is rather an element that accepts an input value upon the (rising) edge of a control signal, and which retains the stored value before and after that transition. There exist various implementations of such elements, the most common one, at least in CMOS technology, being the *master-slave* latch pair. It consists of two latches connected in series, the second latch being controlled by the inverse enable of the first latch. The master-slave register is displayed in Fig. 3.6.

*Master-slave latch pair*

*Clock signal*
*Level-sensitive*
*Edge-sensitive*

Whereas the output of latches depends on their inputs as soon as the control signal G is high, the output of registers changes only upon the rising edge of their *clock* signal. Latches are level-sensitive, registers *edge-sensitive.*

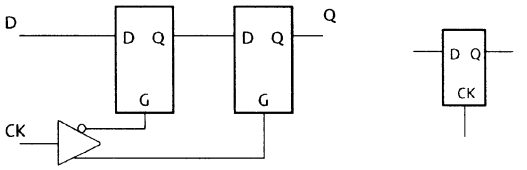


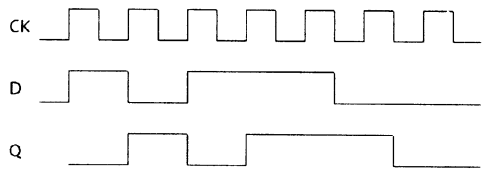

*Fig. 3.6. Master-slave register*

The functioning of this pair is explained as follows: Assume that the control signal CK is low. Then the first latch, the master, is transparent, while the slave holds a value Q. As soon as CK switches to high, the master latches and holds the input value D applied in this moment. The slave becomes transparent and yields the value held by the master at its output. When the control signal CK returns to low, the slave closes again (without a change of Q), and the master becomes transparent. The effect is a total separation of input D and output Q, with Q acquiring the momentary value of D at each rising edge of CK. Typically, CK is a periodic signal, therefore called the *clock.* Such a pair of latches constitutes an *edge-triggered D-register.* Its symbol is displayed in Fig. 3.6, and Fig. 3.7 shows that a register causes its output

*Separation of D and Q*

to be the same as its input delayed by a clock period, expressed by $Q^+ = D$ (or $Q = REG(D)$).

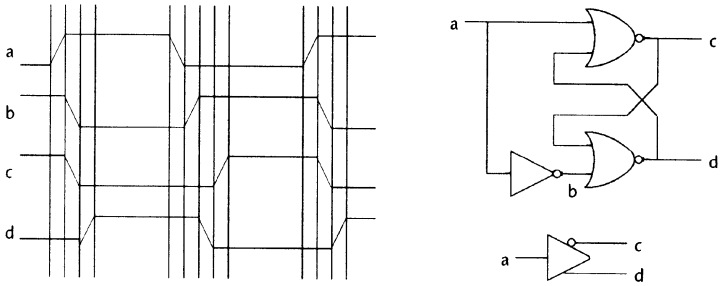
*Fig. 3.7. Output of register equals input delayed by clock period*




If we study the D-register consisting of two latches in series more carefully, we realize that their respective enable signals must never be active at the same time in order to guarantee the promised register nontransparency. The signals must be strictly *non-overlapping*. This property cannot be guaranteed if a simple inverter is used to generate one enable signal from the other. A circuit generating non-overlapping signals c and d is shown in Fig. 3.8 together with the signals as functions of time. The distance between vertical lines denotes a gate delay. Circuits using registers built of latch-pairs are said to be based on a *two-phase clock scheme.*

*Two-phase clock scheme*

*Fig. 3.8. 2-phase clock generator and symbol*




### Direct Set and Reset

There exist components (chips) representing registers with additional input signals (as well as the inverted output $Q' = \sim Q$). The additional inputs are the set and reset lines of the included SR-latch, called *Set* and *Reset* signals. It is important to note that these direct signals override the CK and D inputs, i.e. perform the specified action regardless of D and CK. Because they can cause a change of Q at any time, and not only upon the rising edge of the clock CK, they are also called

*Set*
*Reset*

*asynchronous set and reset.* The symbol used is shown in Fig. 3.9.

*Asynchronous set and reset*

As far as timing is concerned, it suffices to take into account two values only, which are indicated in every data book on standard parts, namely the register's *setup time* and *hold time.* The former is the time prior to the rising clock edge during which the data input must be held constant. The latter is the time after the edge during which the input must not be changed. Typically, these times lie in the order of a gate delay of parts fabricated with the same technology. Their sum determines the highest frequency with which the register may be clocked.

*Setup time*
*Hold time*

$$f_{max} = 1 / (t_{setup} + t_{hold})$$

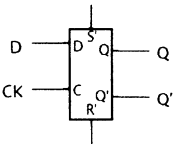



*Fig. 3.9. D-Register with asynchronous, direct Set and Reset*

### Clock Enabling

Many times it turns out to be necessary to trigger a register only at selected edges of a continuous clock train. In this case the clock applied to the register must be derived from the continuous clock and a so-called *enable* signal e'. The simplest way to achieve this is to use a gate which generates a *qualified clock,* having a rising edge only at the end of the clock period during which e' is active (see Fig. 3.10).

*Enable* signal
*Qualified clock*

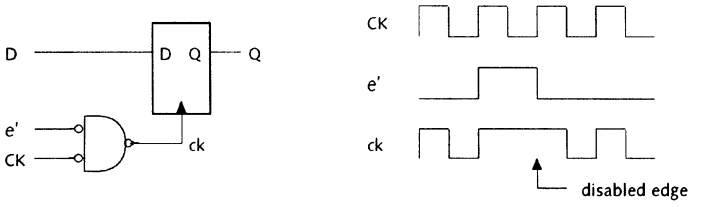



*Fig. 3.10. Clock qualified by enable signal e'*

This very simple solution is at least problematic, because the gate inflicts a propagation delay onto the clock of this register. Such details may unnecessarily complicate the timing analysis task. The preferred solution is the one shown in Fig. 3.11, which conditions the data input instead of the clock. If e' is high, the clock edge causes the register to be loaded with the value previously held instead of the input D. We denote a register with enable signal e', input D and output Q by Q = REG(e', D).

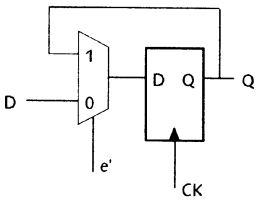



*Fig. 3.11. Register qualified by multiplexer with feedback*

## 3.4. The JK-Register

The JK-register, also called the JK-flipflop, is a variant of the D-register. Instead of a single input D it features two inputs, namely J for setting the register and K for resetting it. Both actions take place at the rising edge of the clock. Like the D-register, the JK-register can be implemented using the master-slave scheme with latches. Additionally, a feedback of the output Q is used to qualify the J and K signals. The circuit is shown in Fig. 3.12. Note that if both J and K are active, the register toggles.

| J | K | Q+ |
|---|---|---|
| 0 | 0 | Q |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | ~Q |

Q+ = J*~Q + ~K*Q

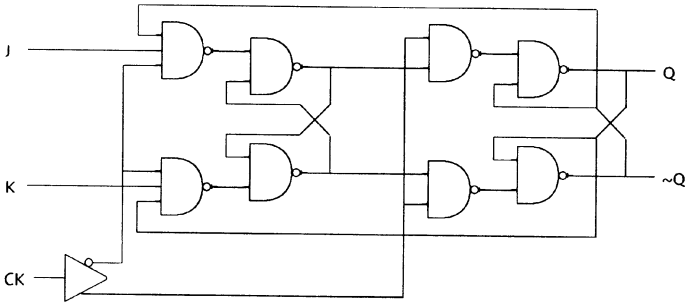



*Fig. 3.12. JK-Register implemented using master-slave latches*

**Summary**

Latches and registers have two stable states and are therefore capable of storing a single bit. The latch is the simpler of the two; it accepts an input value while its control input has the value 1, or, more precisely, is above a certain voltage level. The latch is called a level-sensitive device. The register, on the other hand, accepts an input value at the moment when its control input passes a certain voltage threshold. It is therefore called an edge-sensitive device.

Both level-sensitive and edge-sensitive devices come in two varieties, one featuring distinct inputs for setting and resetting the state, and one with a data input directly indicating the value to be stored. The former variants are the SR-latch and the JK-register, the latter are the D-latch and the D-register.

# Synchronous, Sequential Circuits 4.

**Overview**

Sequential circuits operate in steps. They consist of combinational circuits and registers; the latter represent a state. All registers use the same control signal - the clock - which determines when a new state is assumed. Hence, the sequential circuit is synchronous, and it is called a state machine.

Similar to the class of combinational circuits, there exist frequently used patterns of sequential circuits. If the combinational component is an incrementer, the state machine is a counter. If the combinational part is a multiplexer with inputs from "neighbouring" elements, the result is a shift register.

But state machines are a far more general class of circuits. Their properties or behaviour in time is usually specified in terms of a program or flow diagram. A general method to derive a state machine from such a specification is presented.

Combinational circuits yield their results all at the same time, i.e., to use modern jargon, in parallel. Disregarding gate delays, the results appear instantaneously. Apart from relatively simple tasks, trying to obtain the desired results by mere combinational circuits is prohibitive. The much more economical solution is to approach the results in steps. The results of each step are based on values obtained in the preceding step. Evidently, a facility is needed to retain values from one step to the next. The means to do this have been introduced in the preceding chapter: the register.

*Sequential circuit*

Such circuits are called *sequential,* and they consist of combinational circuits and registers. Because registers have a *state,* so does a sequential circuit. It is called sequential, because the sequence of states in time is of primary interest. As the states change with time, one sometimes speaks of a circuit as running, whereas a combinational circuit's behaviour in time is of little interest (apart from propagation delays).

*Synchronous circuit*

If all registers are edge-triggered and connected to the same clock source, we speak of a *synchronous* circuit. It has many advantages

over asynchronous circuits, and in particular drastically simplifies the timing analysis required to guarantee a proper "running". We will therefore also omit the explicit drawing of clock signals in diagrams, assuming that all registers are clocked by the same, global signal originating from an oscillator. The register *reset* signal, which is used to force the entire circuit into an *initial state*, is also a global signal.

## 4.1. The State Machine

Synchronous, sequential circuits can effectively be represented by a simple scheme called *state machine* (SM). Often the adjective *finite* is added, but since all implementable circuits are finite, it seems somewhat redundant, and we will omit it in this text. A synonym is the *finite automaton*. Its formulation was postulated by R. Moore; the *Moore-type* SM is shown in Fig. 4.1.

*Finite automaton*

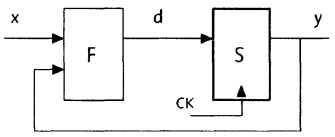



*Fig. 4.1. Moore-type state machine*

Here F is a combinational circuit and S a set of registers. The state y to be assumed upon the next clock cycle depends on the previously held state and the input x. F is called the *next state function*. If we denote the value of y during the *next* cycle by $y^+$, then

*Next state function*

$$y^+ = d = F(x, y)$$

We assume that there also exists a reset signal causing the circuit to enter a specified initial state. Noting that a machine with n registers can assume $2^n$ different states, we realize that its sequence of states must have a *period* of at most $2^n$ cycles (clock ticks).

A slightly more general form of state machine was postulated by G. Mealy (see Fig. 4.2). It adds to the Moore machine a second combinational circuit G, defining the output y as a function of the state s and the input. This circuit is called a *Mealy-type* state machine, and the outputs of registers are called *state variables*.

*State variables*

$$s^+ = d = F(x, s) \qquad y = G(x, s)$$

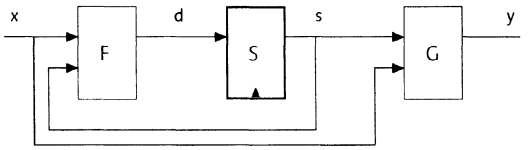



*Fig. 4.2. Mealy-type state machine*

As a first example, let us postulate a Moore-type state machine with states specified by the following tabulation and with a period of 3 clock cycles:

| cycle | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| y0 | 1 | 0 | 0 | 1 | ... |
| y1 | 0 | 1 | 0 | 0 | ... |
| y2 | 0 | 0 | 1 | 0 | ... |

Evidently $y1^+ = y0$, $y2^+ = y1$, and $y0^+ = y2$. The resulting circuit is shown in Fig. 4.3. We realize that a state machine's behaviour can be described by a sequence of values for each variable, discarding actual timing considerations. Real-time figures expressed in seconds do not occur. This is an advantage not to be underestimated.

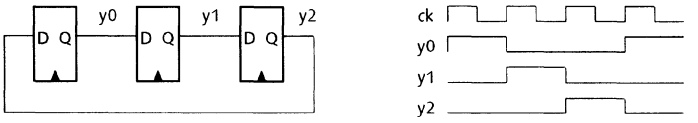



*Fig. 4.3. State machine with period 3 and initial state 1,0,0*

This is, of course, about the simplest state machine conceivable. First of all, it has no inputs, i.e. continuously repeats the same sequence of 3 states. Second, it contains the same number of registers as states assumed. This kind of simple state machine, where in every state exactly one register holds the value 1 is called a *one-hot* state machine. The example turns out to be a frequency divider (by 3).

*One-hot state machine*

It is tempting to simplify this circuit, simplify in the sense of reducing its number of registers. Clearly, for a cycle of 3, two registers suffice, if we encode the outputs. Let the states be defined as follows:

| cycle | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| y0 | 0 | 1 | 0 | 0 | ... |
| y1 | 0 | 0 | 1 | 0 | ... |

We find that $y1^+ = y0$, and $y0^+ = \sim y0 * \sim y1$. The corresponding circuit is given in Fig. 4.4.

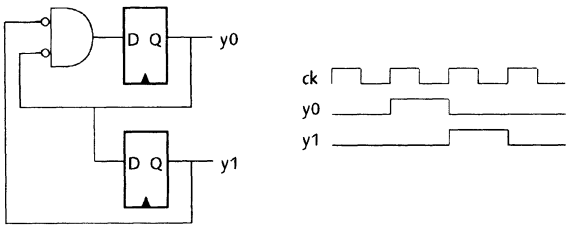



*Fig. 4.4. Simplified state machine with period 3 and initial state 0, 0*

We must note that the term "simplified" here concerns the number of registers. This, of course, is a narrow view. After all, the next-state function now has become more complicated, since it requires a *nor* gate. Much research has been conducted on simplification of state machines in terms of reducing the number of registers. Modern technology, however, does not necessarily let this number emerge as the most significant factor. Sometimes a one-hot solution may be preferable.

In summary, we repeat that synchronous, sequential circuits consist of combinational parts and registers. The registers all use the same clock and reset signals which are considered as global and implied.

## 4.2. The Shift Register

Our second example of a state machine is a shift register. It is obtained by connecting the output of each element to the input of its neighbour, in fact like in the "one-hot" example. If the inputs are fed through a multiplexer with inputs from both neighbouring register elements, we obtain a *universal shift register.* The multiplexer constitutes the combinational circuit representing the next state function F (see Fig. 4.5).

*Universal shift register*

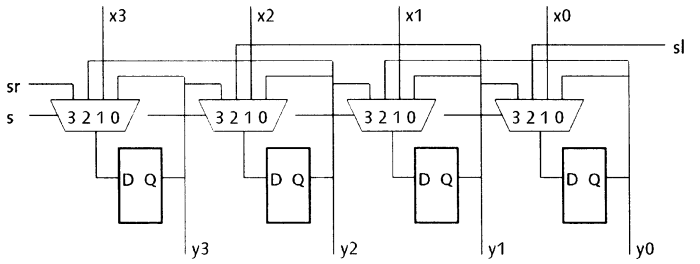



*Fig. 4.5. Universal 4-bit shift register*

The function of the shift register is determined by the selector signal s as follows:

$y_i^+ = MUX(s: y_i, x_i, y_{i-1}, y_{i+1})$

| mode | next state | function |
|---|---|---|
| s = 0 | $y_i$ | hold |
| s = 1 | $x_i$ | load |
| s = 2 | $y_{i-1}$ | shift left (up) |
| s = 3 | $y_{i+1}$ | shift right (down) |

## 4.3. The Synchronous Binary Counter

One of the most frequent circuits is the (binary) counter. We obtain it as our third example of a state machine. The next state must represent the old state incremented by 1 when considered as the binary encoding of a number. The next state function is evidently an incrementer (see Sect. 2.6). The circuit is given by Fig. 4.6.

$y_i^+ = y_i - c_{i-1} \quad c_i = y_i * c_{i-1}$

Assuming en = 1, the counter's state matrix is

| cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| y0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | ... |
| y1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | ... |
| y2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ... |
| c2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | ... |

and if en = 0, the counter retains its state, i.e. doesn't count. *en* is therefore called the *enable input.*

*Enable input*

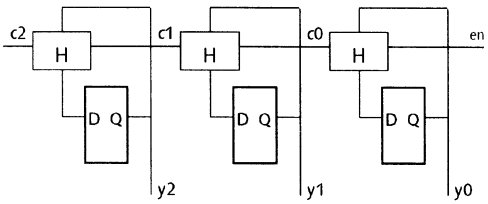



*Fig. 4.6. 3-bit binary counter*

*Synchronous clear*

A minimal and indispensible addition is a control signal to reset the counter to an initial state, typically zero. Figure 4.7 shows two ways to implement the clear function for a counter element. We point out that this signal is called *synchronous*, because it takes effect upon the next clock edge, in contrast to an asynchronous clear as encountered in Chapter 3 (Fig. 3.9), which changes a register's value disregarding the clock. Note that in the version to the right in Fig. 4.7, *cin* must be active (1), if *clr* is to be effective.

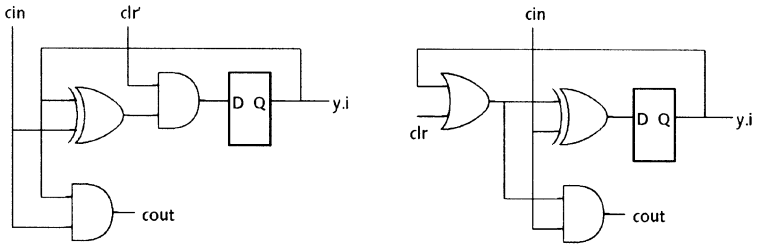



*Fig. 4.7. Counter elements with clear signal*

Commercial components are available as simple counters and also as up/down counters. In this case, the feedback from the register output to the half adder must be, upon selection of down counting, inverted. Typically, counters are provided with a clear signal (initial state = 0), or a load signal together with data inputs, or both. A single stage of such an up/down counter is specified in detail in Fig. 4.8. Again, we easily recognize the state machine as the underlying model, here with the next state function

$$y_i+ = \text{MUX}(ld': x_i, y_i - up' - cin) * clr'$$

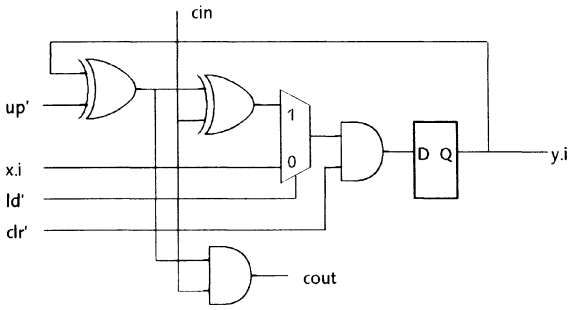



*Fig. 4.8. Up/down counter element with synchronous clear and load*

## 4.4. A Design Methodology for State Machines

After these examples of standard cases of state machines, we proceed to the task of designing state machines according to requirements of given applications, i.e. to custom-tailored state machines. The rules to be obeyed in this design process are called a methodology. The first question arising is: "How are state machines specified?" Because they exhibit a behaviour, i.e. change of states in discrete time intervals, they are naturally described by a program. Since state machines are in most cases quite simple, i.e. with a dozen states or less, such programs have traditionally been specified in terms of flow diagrams.

*Flow diagrams*

The flow diagrams consist of nodes (boxes), each representing a distinct state and labelled with a name. In each box, the values of the output signals holding in the respective state are specified. The directed arcs between the nodes represent state transitions. Arcs are labelled by the condition under which the transition occurs. If an arc is unlabelled, the transition is unconditional. We may imagine a token being placed in some box, marking the current state. The token proceeds to another box indicated by an outgoing arc upon each clock cycle. An example is shown in Fig. 4.9. There are 4 states, an input signal x, and an output signal y.

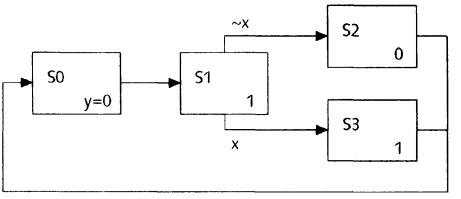



*Fig. 4.9. Flow diagram specifying state machine*

An equivalent, tabular representation of the state machine of Fig. 4.9. is the following:

| state | output y | input x | → | next state |
|---|---|---|---|---|
| S0 | 0 | - | → | S1 |
| S1 | 1 | 0 | → | S2 |
| S1 | 1 | 1 | → | S3 |
| S2 | 0 | - | → | S0 |
| S3 | 1 | - | → | S0 |

*Completeness*

Before embarking on an implementation, its specification must be checked for completeness and determinism. A specification is *complete,* if for each state a next state is always defined, i.e. the conditions $x_i$ of the outgoing arcs form a complete covering: $x_0 + x_1 + ... + x_{n-1} = 1$.

*Determinism*

A specification is *deterministic,* if for each state the conditions of its outgoing arcs are mutually exclusive, i.e. $x_i * x_j = 0$, for all i different from j.

*One-hot SM*

Then we need to decide what kind of implementation is to be made. If registers are easily available, the *one-hot* machine is the preferred solution; this is for example the case when using FPGAs (field programmable gate arrays). If registers are relatively scarce in comparison with gates for decoding circuitry, an *encoded* version is preferable. This is mostly the case when using discrete components or PLDs (Sect. 4.5).

*encoded SM*

The derivation of a one-hot implementation is quite straight forward. Each box (state) is represented by a register. The input of each register is the logical conjunction of the terms represented by the incoming arcs. The terms are the logical disjunction of the state signal and the condition under which the transition is taken (see Fig. 4.10).

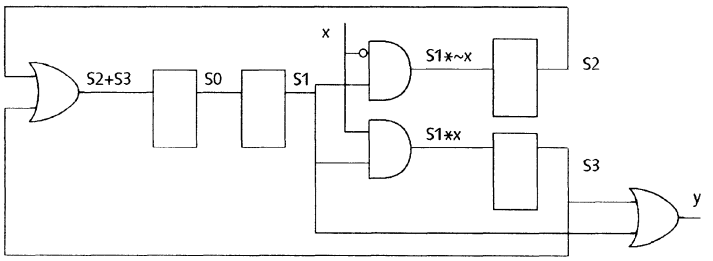



*Fig. 4.10. One-hot machine version of the specification of Fig. 4.9*

If we decide to build a machine with state encoding, the first step is to determine the number of registers and to select the encoding of states in terms of register values. This is called *state assignment* or *state encoding*. It is advantageous to choose the values so that the minimum number of register values (bits) change between adjacent states. This usually minimizes the complexity of the circuitry needed to implement the register inputs. For our example with 4 states, 2 registers suffice, and we choose the register values q1 and q0 as the binary representation of the state number.

*State assignment*
*State encoding*

In order to determine the input $d_i$ of a register $q_i$, we tabulate all state transitions. From the table it is easy to derive the corresponding register input expressions:

| state (q1, q0) | input (x) | → | next state (d1, d0) |
|---|---|---|---|
| 0 0 | - | → | 0 1 |
| 0 1 | 0 | → | 1 0 |
| 0 1 | 1 | → | 1 1 |
| 1 0 | - | → | 0 0 |
| 1 1 | - | → | 0 0 |

from which we see immediately that

$$d0 = (\sim q1 * \sim q0) + (\sim q1 * q0 * x) = \sim q1 * (\sim q0 + x)$$
$$d1 = \sim q1 * q0$$

The output y is 1 in states S1 and S3, and is therefore given by the expression

$$y = \sim q1 * q0 + q1 * q0 = q0$$

and the resulting circuit is shown by the schematic of Fig. 4.11.

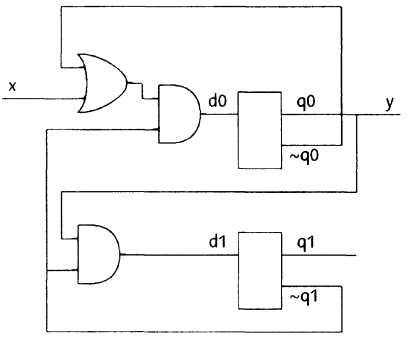



*Fig. 4.11. Encoded version of the specification of Fig. 4.9*

In general, it is not easy to find an optimal encoding which minimizes the number of gates needed, and therefore several iterations are needed for evaluating various state assignments. Sometimes, mixtures of one-hot and encoded solutions are employed, depending on the resources (technologies) available.

We now show the implementation of a second, slightly more complex example, a state machine M whose properties are specified verbally as follows:

The circuit has 3 inputs x0, x1, and x2, and 3 outputs y0, y1, and y2. It remains in its so-called *idle* state (all outputs low) until x0 goes high. Then the *start* state is entered and y0 goes high for a single cycle. If x1 is low, the machine enters the *active* state for one cycle, in which y1 is high. Otherwise a single cycle with all outputs low follows, and then y1 goes high until x2 becomes high, whereafter the active state is entered. The active state is always followed by a single cycle *end* state with y2 high; then the machine returns to its idle state.

This specification is expressed in terms of the flow diagram in Fig. 4.12.

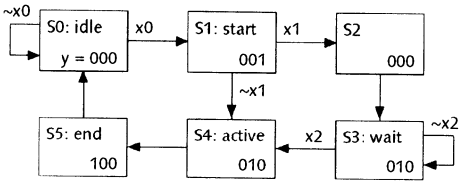



*Fig. 4.12. Flow diagram of state machine M*

As in the first example, the derivation of a one-hot solution is straight forward. It is shown in Fig. 4.13. Note that the specification is both complete and deterministic. The register input expressions are derived directly from the flow diagram:

$d0 = (q0 * \sim x0) + q5$ $\quad$ $d3 = q2 + (q3 * \sim x2)$

$d1 = q0 * x0$ $\quad$ $d4 = (q1 * \sim x1) + (q3 * x2)$

$d2 = q1 * x1$ $\quad$ $d5 = q4$

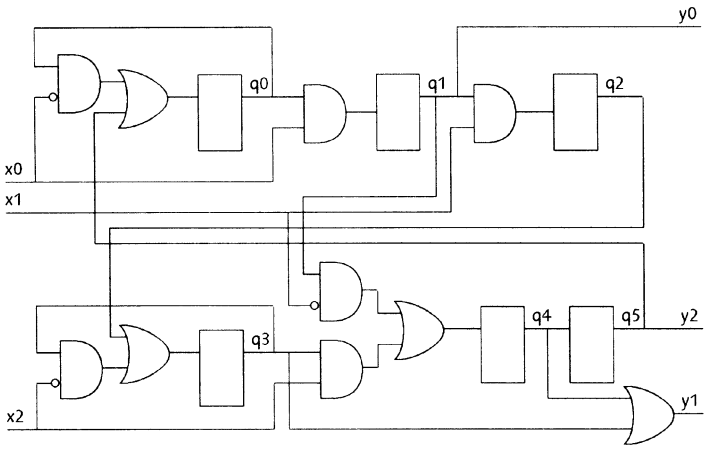



*Fig. 4.13. One-hot solution for state machine M*

The first step in the derivation of the alternative, encoded implementation, is the state assignment. The following proposal leads to a reasonably good solution:

S0 q = 000
S1 q = 001
S2 q = 101
S3 q = 111
S4 q = 011
S5 q = 010

The state transistions are summarized by the following table:

| state | (q2,q1,q0) | input (x2,x1,x0) | → | next state | (d2,d1,d0) |
|---|---|---|---|---|---|
| S0 | 000 | --0 | → | S0 | 000 |
| | 000 | --1 | → | S1 | 001 |
| S1 | 001 | -0- | → | S4 | 011 |
| | 001 | -1- | → | S2 | 101 |
| S2 | 101 | --- | → | S3 | 111 |
| S3 | 111 | 0-- | → | S3 | 111 |
| | 111 | 1-- | → | S4 | 011 |
| S4 | 011 | --- | → | S5 | 010 |
| S5 | 010 | --- | → | S0 | 000 |

Expressions for the register inputs d are readily derived from this table

$$d0 = (\sim q2 * \sim q1 * \sim q0 * x0) + (\sim q2 * \sim q1 * q0) + (q2 * q0)$$
$$d1 = (\sim q2 * \sim q1 * q0 * \sim x1) + (q2 * q0) + (q1 * q0)$$
$$d2 = (\sim q2 * \sim q1 * q0 * x1) + (q2 * \sim q1 * q0) + (q2 * q1 * q0 * \sim x2)$$

and the outputs follow from their definition in the state diagram

$$y0 = \sim q2 * \sim q1 * q0$$
$$y1 = (q2 * q1 * q0) + (\sim q2 * q1 * q0) = q1 * q0$$
$$y2 = \sim q2 * q1 * \sim q0$$

We should point out that not all possible values of the registers represent a state, i.e. there exist "illegal states". Care has to be taken that if such an illegal state is assumed - for example when the circuit is switched on (powered up), the next transition leads to a legal state. Typically, however, registers are used which feature a reset signal; then the state where all register values are 0 is distinguished and is chosen as the legal initial state.

## 4.5. The PLD and the FPGA with Registers

*PLD with registers*

The tabular representation of Boolean functions in the preceding example of state machine M strongly suggests the use of PLDs for their implementation. Fortunately, there are PLDs available that not only consist of the *and* and *or* matrices for combinational functions (Sect. 2.10), but in addition feature a set of registers. Feedback paths from the register output to the *and* matrix make them ideally suited for the realization of state machines, as shown in Fig. 4.14.

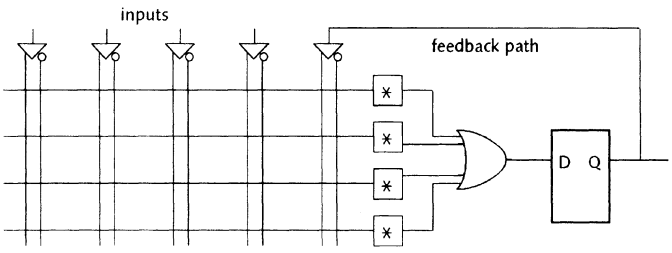



*Fig. 4.14. Section of PLD with register*

Modern PLDs feature a so-called *macro cell* which includes a register and further, programmable multiplexers (see Fig. 4.15). Appropriate output selection permits the particular section to be used as a combinational or a registered circuit, and with either straight or inverted output. The feedback selection makes it possible to use either the term or the registered value as a factor in product lines. Furthermore, by placing a tri-state gate (see Sect. 5.3) between the register and the pin, the output can be disconnected from the pin, which may therefore be used as an additional input to the product lines.

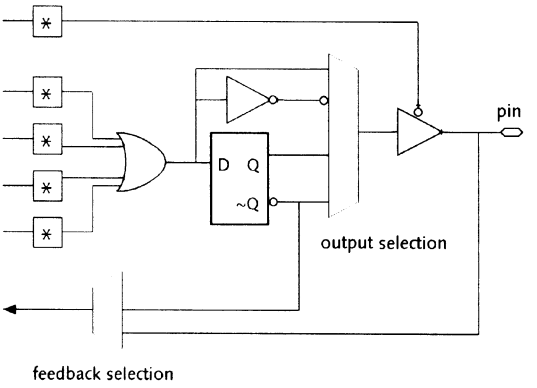



*Fig. 4.15. Typical, programmable macro cell of PLD*

Programmable gate arrays (FPGAs, see Sect. 2.11) typically contain a register in every cell. A cell's output may thus be selected as that of the cell's combinational function or of its registered value. An additional feedback path lets the cell become a register with enable or a D-latch. The a-multiplexer selects y (or ~y), acting as data input d, while the b-multiplexer is set to f. Hence, x acts as register enable or as latch control. Figure 4.16 shows an example of such a cell derived from that of Fig. 2.26.

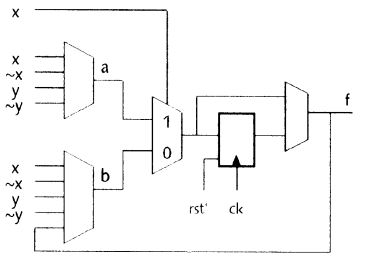



*Fig. 4.16. FPGA cell with D-register*

## 4.6. Timing and Practical Considerations

So far we have virtually ignored timing considerations. They are indeed reasonably simple in the case of synchronous circuits, where all registers are controlled by the same clock signal. The clock period must simply not be shorter than the time a signal needs to progress from a register's output through the combinational circuit to a register's input. The critical entry is the *longest path* of all these signals in the entire state machine. To its propagation time we must add the set-up time of the destination register. The maximum clock frequency is

*Longest path*
*Maximum clock frequency*

$$f_{max} = 1 / (t_{pmax} + t_{setup} + t_{hold})$$

A synchronous circuit can be operated as fast as the longest signal path in its combinational part will allow, but not faster. Equilibrating path lengths, i.e. avoiding the case where a few paths (even a single path) are substantially longer than most others, is an essential task for the professional designer.

A second constraint concerns the minimal propagation delay of the combinational logic. The registers' minimum propagation time plus the minimal length of all combinational paths must be greater than the destination registers' hold time. Otherwise the inputs change too fast.

This yields a useful hint for finding errors. If a circuit doesn't function at any frequency, it is likely that the cause is too long a hold time. On the other hand, if the circuit functions properly up to a certain clock rate, and fails for higher rates, the cause is probably too large a propagation delay.

Synchronous circuits use the same clock for all registers. It is absolutely essential for proper operation that the clock edges occur at the same time everywhere. In implementations, clocks are fed through wires, and since they have a certain length, resistance, and parasitic capacitance, delays may occur. They cannot be ignored when high clock frequencies are used. Obtaining minimal (not zero!) time differences (*clock skew*) is one of the hard practical problems in high-speed circuit construction.

*Clock skew*

Quantitatively, clock skew is the maximum difference in the clock's arrival time at any flipflop. It is always positive. For a clock system to

work properly, the worst-case clock skew must be included in the expressions for the maximum frequency and the hold time condition as follows:

$$f_{max} = 1 / (t_{pmax} + t_{setup} + t_{skew})$$
$$t_{pmin} + t_{logicMin} + t_{skew} < t_{hold}$$

Of equal importance is that clock signals are not being distorted, i.e. display sharp edges everywhere. The combination of long distribution lines and high frequency often leads to such problems. When building a circuit, the inspection of proper clock signals should follow immediately after inspection of proper voltage supply connections. Knowledge about signal propagation and line transmission is indispensible for mastering critical situations.

**Summary**

A sequential circuit consists of combinational circuits and registers, all controlled by a common, periodic signal called clock. The registers represent the so-called state. The registers' outputs, together with external signals, are the inputs of the combinational circuits, whose outputs are fed to the registers and hence represent the state in the next clock cycle. Thus, the next state is a function of external signals and the current state. Since the state machine assumes a sequence of states in time, it is called a sequential circuit.

State machines are at the heart of almost all digital devices, in particular computers. Their analysis is relatively simple, as their behaviour can be represented by sequences of discrete values, one value for each clock cycle. Circuit analysis becomes infinitely more complex as soon as registers are clocked asynchronously.

Particularly attractive for the implementation of state machines are programmable devices (PLDs) which, in addition to the general combinational circuits, feature registers and feedback paths from the register outputs to the inputs of the combinational parts. They are called register-PLDs.

# Bus Systems

# 5.

**Overview**

In all circuits presented so far, all connections between gates are uni-directional. Every output of a component is connected to inputs of other components only. In contrast to this stands the bus. It is capable of connecting several outputs among which, however, only one may be active at any instance in time. Bus systems are used to reduce the number of wires connecting units of a system quite drastically. This is particularly desirable in the case of larger units and hence longer wires.

## 5.1. The Concept of a Bus

Computers typically consist of various units. They are not only units in a functional or logical sense, but often also physical units, connected by wires of some length. Each component (chip) forms a unit; the wires connecting the chips are very much longer than their internal connections. So are the individual boards in a computer; they are linked through connectors that make the boards exchangable. Hence, the concept of units is to be understood as being hierarchical.

A computer, for example, consists of a control unit, an arithmetic unit, memories, and (interfaces to) peripheral devices. Let us now assume that we wish to make it possible that each unit may transfer data to any other unit. The obvious solution consists of a multiplexer associated with each unit (see Fig. 5.1).

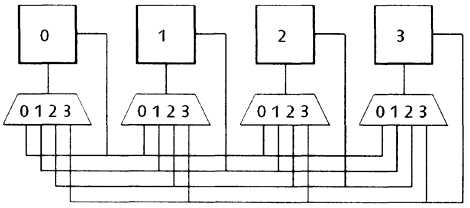
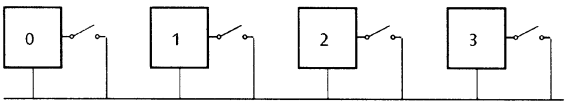



*Fig. 5.1. Strongly connected system with 4 units*

The obviously undesirable property of this solution is the large number of connecting wires required, particularly considering that these wires are linking units physically apart, and are therefore relatively long. A more economical solution can only be found by making

some concession. The concession to be offered is that at any time only one pair of units may communicate, or, relaxing this restriction, that at any time at most one unit may act as a data source. This concession, which in the majority of applications is easily acceptable, leads to a system of units connected by a so-called *bus* as shown in Fig. 5.2.

*Fig. 5.2. Units connected by a bus*



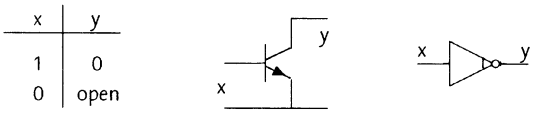

Because at any time at most one unit may be connected as a data source, it is necessary that the other units can be disconnected. Hence, a switch becomes necessary, an element not encountered so far in this book. Can a switch be realized with components similar to gates? Two solutions to this problem are in wide use, the open-collector and the tri-state circuits.

## 5.2. The Open-Collector Circuit

An *open-collector* gate is a circuit representing a gate whose output is not a proper function: the collector of the output transistor is left undefined (open), and it is directly connected to a pin. The most elementary open-collector circuit is the oc-inverter (see Fig. 5.3).

*Oc-inverter*

*Fig. 5.3. Open-collector function, circuit, and symbol*




In contrast to a normal inverter, this circuit is capable of generating a 0, but not a 1. The term "open" means that the transistor does not define the value; if it is to be 1, this must be defined elsewhere, typically by a (external) pull-up resistor R. It defines the bus to be high by default, with every connected gate being able to "pull down". This leads to the so-called open-collector bus as shown in Fig. 5.4. Inverters are replaced by *nand* gates with one input playing the role of the selector s (effectively controlling the switch of Fig. 5.2.) and the other being the data input x. Note that the data on the bus are the *inverted* values of

*Pull-up resistor*

*Open-collector bus*

the sources, and that there are no central elements apart from the passive pullup resistor.

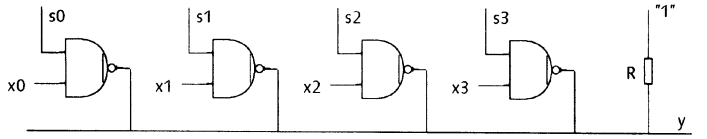



*Fig. 5.4. Open-collector bus*

Here we have assumed that at any moment at most one selector signal is active, namely the one designating the current data source. Independent of this restriction, the bus signal y is given by the expression

$$y = \sim(s_0x_0 + s_1x_1 + ... + s_{n-1}x_{n-1})$$

The open-collector bus is therefore also called a *wired-or* circuit; the *or* gate of this multiplexer has degenerated into the bus wire.

*Wired-or circuit*

The open-collector method is nowadays employed only rarely to connect larger units of a computer. This is because it has a significant handicap explained as follows, referring to Fig. 5.5:

Upon an input transition from low to high, the transistor turns on and the parasitic capacitance (which is relatively high for long bus wires) is quickly discharged through the low internal resistance of the transistor. If, however, the inverse transition occurs, the capacitance is charged through the central pullup resistor. Because this resistor cannot be made very small, as it would present an overload to the pull-down transistors, the time for loading is large compared to the time of discharge. As a consequence, open-collector circuits are relatively slow.

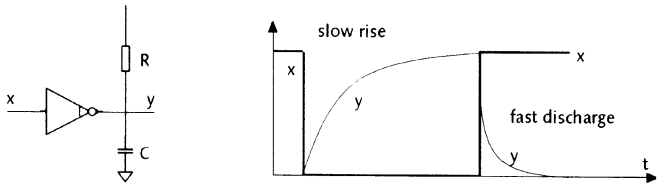



*Fig. 5.5. Open-collector output y as function of x in time*

The open-collector circuit is, however, ubiquitous in highly integrated circuits such as PLDs, ROMs, and memories, where wires are short and capacitances small. We recall that ROMs consist of an *and* and an

*or* matrix, with the latter being programmable. The former consists of rows which constitute product lines. They are implemented as *wired and gates* according to the formula

$$y = \sim(x_0 + x_1 + ... + x_{n-1}) = \sim x_0 * \sim x_1 * ... * \sim x_{n-1}$$

The circuit takes the form of two matrices with each node containing a single transistor (see Fig. 5.6). The outputs of the *and* matrix (vertical lines) represent the cell selection signals.

$$s_0 = \sim x_1 * \sim x_0 \quad s_1 = \sim x_1 * x_0 \quad s_2 = x_1 * \sim x_0 \quad s_3 = x_1 * x_0$$

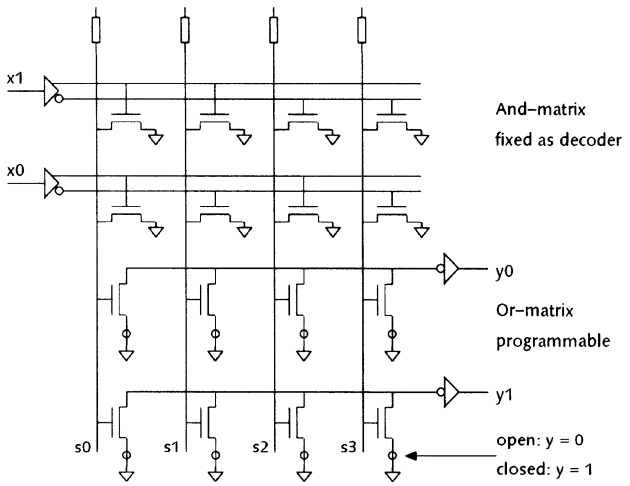



*Fig. 5.6. 4 x 2 bit ROM implemented by transistor matrices*

Apparently it is the open-collector technique which makes the high component density characteristic for ROMs and PLDs possible. We now realize that the abbreviating notation for product lines and their *and* gates mirrors quite closely their implementation with the open-collector technique.

## 5.3. The Tri-state Gate

As mentioned above, the open-collector circuit suffers from speed deficiency. The preferable solution is the tri-state gate. A *tri-state* gate has 3 possible output states: low (0), high (1), and open. Effectively, the tri-state gate represents a driver followed by a switch (see Fig. 5.7).

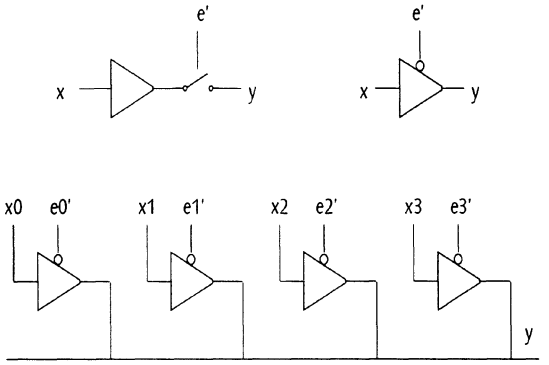



*Fig. 5.7. Function and symbol for tri-state gate; tri-state bus*

The tri-state bus has no central elements at all. The outputs of all gates are connected together. In contrast to the open-collector bus, it is important that never more than a single gate is enabled at the same time. If two gates were enabled, one of them might drive the bus low, while the other drives it high, or vice versa. This would constitute a short circuit drawing a high current. A wrongly designed circuit using tri-state gates may not only yield the wrong behaviour, but may even permanently damage the circuitry. Special care is therefore advisable.

The implementation of a tri-state gate turns out to be almost identical to the *nand* gate with totem-pole output stage (see Fig. 1.5). In fact, the only addition is a diode D pulling the base of T4 low. The circuit is shown in Fig. 5.8.

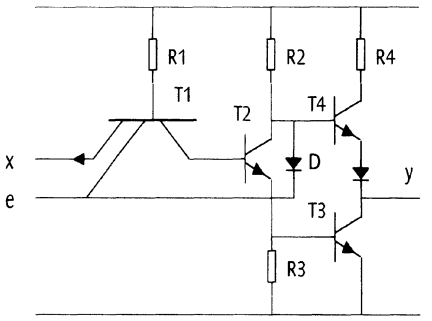



*Fig. 5.8. Circuit of tri-state gate*

If e = 1, the diode is reverse biased (not conducting), and the circuit functions as a *nand* gate, i.e. y = ~x. If e = 0, T1 is conducting and T2

open; therefore T3 is open, and T4 also because the diode pulls its base low. Hence, the output appears as unconnected (floating).

By convention, tri-state enable signals are taken as active-low signals (e'). Also, a typical tri-state gate is a pure switch without implied inversion. A frequently encountered circuit component consists of two tri-state gates connecting x and y in opposite directions. This is called a *transceiver* (see Fig. 5.9).

*Transceiver*

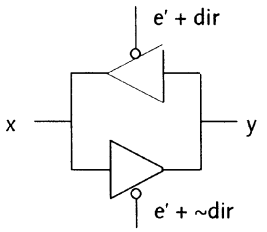



*Fig. 5.9. Transceiver controlled by enable signal e' and direction signal* dir

**Summary**

Bus-systems require drivers that are capable of releasing the driven bus, i.e. drivers that look like uni-directional switches. There exist open-collector and tri-state busses.

An open-collector driver is capable of forcing the driven bus to 0, but not to 1. The bus assumes the value 1 by default, i.e. whenever no driver is active.

The more modern system is the tri-state bus, capable of forcing a bus to assume both 0 and 1 values. The advantage of the tri-state bus is its higher speed, the disadvantage that care must be taken to avoid bus contention, the case where simultaneously active drivers work against each other.

# Memories

# 6.

**Overview**

Memories play a special role among circuit components in the sense that they display a highly regular structure and are therefore particularly suitable for miniaturization and extreme integration. They most spectacularly demonstrate the advances of semiconductor technology: Whereas in 1975 chips with 1K bits represented the state of the art, in 1995 chips with a capacity of 16M bits are available. This is a factor of 214 in 20 years, or a factor of 2 about every 18 months.

We distinguish between read/write and read-only memories. As was explained in Chapter 2, read-only memories (ROM) are purely combinational circuits. An essential property of these memories is that the access time does not depend on the cell selected (addressed). This implied that any cell can be addressed "at random", yielding the same access delay. This property has led to the term "random access memory" (RAM). It stands in contrast to memories where access proceeds through a sequence of "cells", and access time therefore strongly depends on the address value. Such sequential access is typical for all memories with moving parts, such as magnetic tapes and disks. RAMs are electronic memories, also called semiconductor memories, and are on average about 1000 times faster than magnetic memories with moving parts. Here we shall discuss electronic memories only. Among read/write memories one distinguishes further between static (SRAM) and dynamic (DRAM) memories.

## 6.1. Static Memories

For the designer of digital circuits, it may suffice to regard the (static) memory as a device with an address input, a data input, and a data output. In addition there exist control signals, namely CE' (or CS') used for enabling (or selecting) a chip (chip enable / chip select), and WE' (write enable) for indicating a write (store) operation. Often the data inputs and outputs are multiplexed, i.e. the output is internally separated from the external connections (pins) through a tri-state gate. Thereby the number of data pins is halved. In this case, an additional control signal OE' is available which controls the tri-state gate.

*Chip enable/ chip select write enable*

Typically, all these control signals are active-low (see Fig. 6.1).

It must be ensured that the address values are stable while the write-enable signal is active. If this condition is violated, it may happen that data at several addresses are being affected. Furthermore, it is important that the WE' signal itself is "clean", i.e. does not have any glitches.

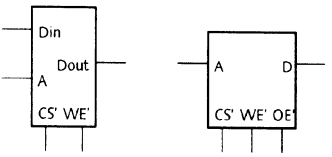



*Fig. 6.1. Static RAMs, multiplexed inputs and out-puts on the right*

*Width*

A RAM with n address pins contains $2^n$ words. If it has m data pins, then each word stores m bits. m is called the *width* of the RAM. Static RAMs are available with m = 1, 4, and - most frequently used - 8. If a memory has to be built with a higher capacity than the chip available, chips are arranged in the form of an array using a separate decoder for enabling different chips. Figure 6.2 displays a memory built with 4 chips with a doubled address range and a doubled width.

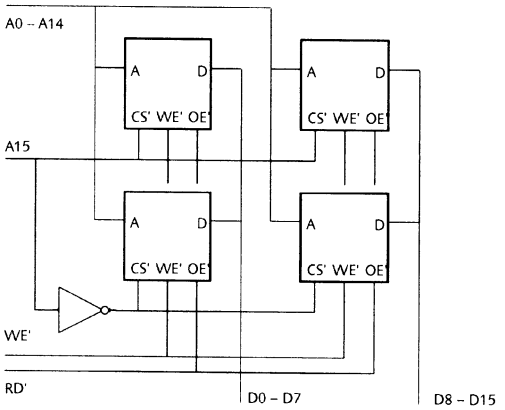



*Fig. 6.2. Memory consisting of 4 RAM chips*

*Access time*
*Cycle time*

The designer also needs to be aware of timing specifications of memory components. The key characteristics are the *access time* and the *cycle time*. The former indicates the delay with which data appear after the select signal becomes active and address signals are stable. The latter indicates the time that must elapse between consecutive accesses. This is illustrated by Fig. 6.3. Typical cycle times for SRAMs are 0.1 µs and less.

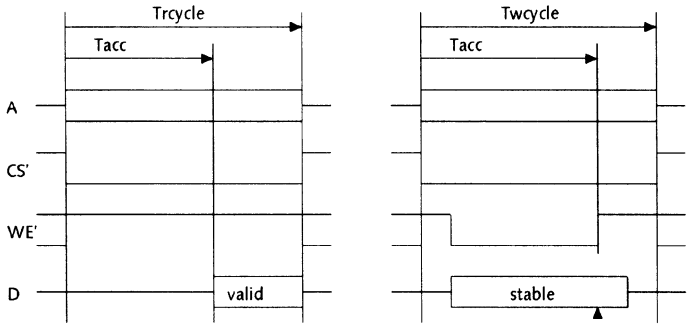



*Fig. 6.3. Dynamic characteristics of RAM*

Knowledge of these properties will suffice in general for a designer. Here we wish to investigate some of the innards of a SRAM, at least we wish to obtain an idea of its structure. Essentially, it consists of an array of cells, each containing a latch. The array is formed as a matrix. Half of the address lines are used to select a row of cells by a decoder. Selection causes all cells of that row to be connected to a (vertical) data line (see Fig. 6.4).

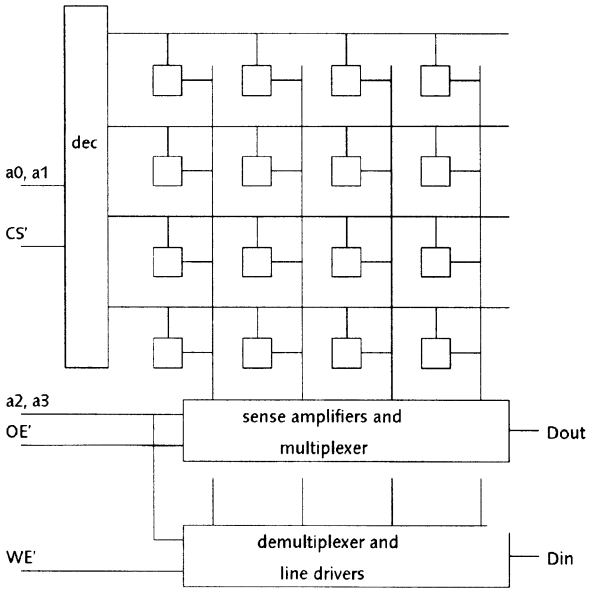



*Fig. 6.4. Structure of 16-cell SRAM*

*pass transistors*

Most modern SRAMs are built in CMOS technology. A cell then consists of 6 FETs which constitute a flipflop as shown in Fig. 6.5. The transistors connected to the data lines are called *pass transistors.* They conduct when the gate is pulled high by the address line. Because such an n-channel pass transistor only conducts zeroes well, each data line is represented by a pair D, D' with D' = ~D. Hence, one of them always carries the value 0. The D-lines represent open-drain circuits, analogous to the open-collector circuits presented in Chapter 5.

*open-drain circuits*

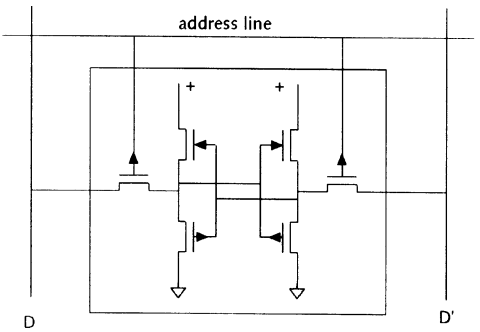



*Fig. 6.5. Six-transistor SRAM cell*

In a read cycle, the selected address line connects the cells of a single row to the data lines. They feed the sense amplifiers. One of them is selected by the multiplexer and fed to the data output pin. A row of cells is sometimes called a *word,* and the address line a word line. We may then call the word line selector the *row address,* and the address feeding the multiplexer the *column address.* We realize that by arranging the cells as a matrix, the multiplexer and the demultiplexer have been drastically simplified.

*Row is called a word*
*Row address*
*Column address*

## 6.2. Dynamic Memories

Memories are the most regular and the densest components designed and fabricated. In order to increase their density even further, the search went for even simpler cells. Since there is no way to implement a flipflop with fewer transistors, another medium for storing data had to be found. The obvious candidate in an electronic circuit is the capacitor, i.e. its charge. The charge and thereby the data are held in a simple capacitor connected by a pass transistor to a data line that

again acts as both a sense line in read cycles and a loading line in write cycles. This truly simple cell is shown in Fig. 6.6.

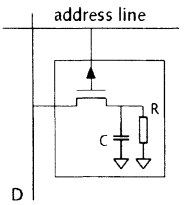



*Fig. 6.6. Single-transistor DRAM cell*

The trouble with all capacitors is that they leak and slowly discharge. Even if the leakage resistance is very high, the decay is significant when the capacitance is small. With the current trend to reduction of device size, the capacitance C of a DRAM cell lies in the order of $10^{-15}$ F, which together with a resistance R of some $10^{12}$ Ω results in a time constant RC in the order of milliseconds. This defect is overcome by regularly reading the stored data and rewriting them, thereby recharging the capacitors. This activity is called *refreshing*. In fact, every read operation draws charge from the capacitor, which therefore must be recharged. However, since it cannot be guaranteed that every cell of a memory chip will be read frequently enough, a refresh process is needed which continuously refreshes all cells, typically row by row at a rate of about 500 cycles/s.

*Refreshing*

The crucial part of a DRAM are its sense amplifiers which must be able to reliably detect relatively small voltage differences. The sense amplifiers use a high degree of positive feedback, which causes them to recharge the read cell automatically.

Dynamic RAM technology has advanced to an astonishing degree. Currently chips with a capacity of 16M bits are on the market, 64M chips are available as samples, and 256M chips are in the development stage. Since at the same time the trend for higher chip density of printed circuit boards continues, smaller chips are desirable, and this in turn requires a small number of pins. In order to address 4M = $2^{22}$ cells, however, 22 address lines are needed. The number of address line pins can be halved if the address is supplied in two parcels one after the other. Since the address is split anyway into a row and a column address, it is only natural to supply the row address first, let the

*Time-multiplexing*

row decoder circuitry settle, and then supply the column address part for selecting the desired data line. This technique is called *time-multiplexing*. It complicates the circuitry required around DRAMs ever further. However, highly integrated, special *DRAM controllers* are available which not only produce the correct timing for the control signals, but also take care of the refresh problems (see Figs. 6.7 and 6.8). The control signals RAS' (row address strobe) and CAS' (column address strobe) indicate which part of the address is fed through the multiplexer to the RAM's address pins.

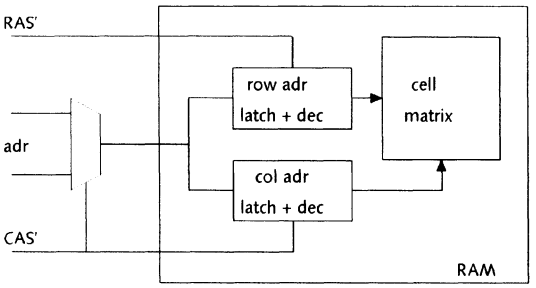



*Fig. 6.7. RAM with time-multiplexed row and column addresses*

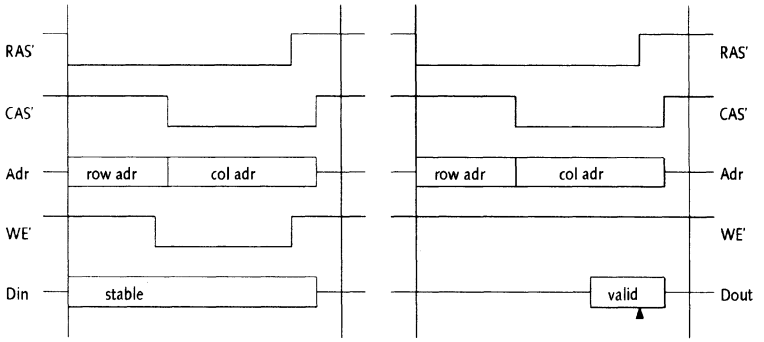



*Fig. 6.8. Write and read cycle of DRAM*

## 6.3. Dual-Port Memories

When a single bit is to be read from a memory organized as a matrix of cells, an entire row is accessed (and refreshed). The addressed bit is then selected by a multiplexer. Evidently, a large amount of data is extracted from the matrix and then discarded. This wasteful treatment of data can be avoided in the cases where it is known that consecutive accesses address consecutive cells (serial access). This is indeed no

exceptional case, and DRAMs with a special provision have been developed which feature a second, serial port. They are called *dual-port memories.*

*Dual-port memories*

Such memories basically function like ordinary RAMs. However, if an additional control signal T is active, the read data are directed into an internal register rather than into the multiplexer. The register has as many elements as are memory cells in a row of the matrix, and it is organized as a shift register. After reading a row onto the shift register, the data are read serially by applying a shift clock. We note that this reading of the shift register occurs completely independently of the other parts of the RAM; the shift register is decoupled except for loading (see Fig. 6.9).

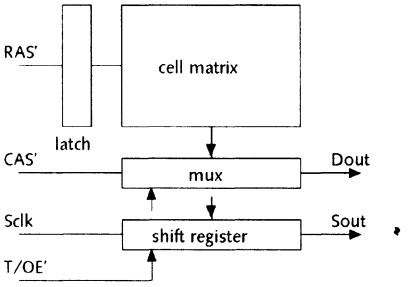



*Fig. 6.9. Dual-port RAM with serial output*

The predominant application of dual-port RAMs is the refresh store for displays. The displayed data are stored in a RAM and must be continuously refreshed. These data are naturally read in strictly serial fashion. The dual-port arrangement provides a highly desirable decoupling of the data access for display refresh from other accesses. Because of this application, which had spurred the development of RAMs with internal shift register, these RAMs are often called *video RAMs.* However, most dual port RAMs allow the shift register to be used also for serial input, and writing an entire row of cells simultaneously.

*Video RAMs*

**Summary**

We distinguish between static and dynamic memories. Both consist of an address decoder, a matrix of storage cells, and a row of sense and write amplifiers. In the static memory (SRAM), every cell is a latch consisting of six transistors. It holds the state permanently. The cell of a dynamic memory (DRAM) consists of a small capacitor holding an electric charge, and of a single transistor acting as a switch. Since capacitors inherently leak, the charge is "dynamic" and has to be re-established periodically.

The advantages of SRAMs are their simplicity in operation and their speed; those of DRAMs are the smallness and low power consumption of their cells, and therefore the high density of cells. For their operation including charge refreshing, special DRAM controllers are used.

# 7. Formal Description of Synchronous Circuits

**Overview**

Traditionally, circuits have been described by diagrams called schematics. They more or less directly mirror the actual layout of components and wires. With growing complexity of circuits, the appropriateness of schematics becomes questionable, and the advantages of textual specifications emerge more distinctly. In texts, structures or patterns can be expressed separately, and thereafter be instantiated and parametrized arbitrarily.

Recently, such formalisms, called hardware description languages (HDL), have been standardized, thereby gaining acceptance in industry. For explanatory and tutorial purposes, however, they are utterly unsatisfactory, surpassing in complexity even large programming languages. Hence, we present our own formalism called Lola, which is confined to express as succinctly as possible the basic and most relevant aspects of digital circuits.

## 7.1. Motivation

In order to describe the various elementary circuits encountered so far, two different forms have been used: *Schematics* using graphical symbols, and *formal texts*. Both have been used, because both have their merits, but also their deficiencies.

Schematics are probably the more popular means to represent circuits in engineering circles. Their evident advantage is that they closely mirror the actual circuit in the sense that lines connecting symbols stand for wires connecting components. In fact, a good schematic provides useful hints for the layout of a circuit on a board. The graphical form also has the advantage of providing a good overview, of presenting "the whole picture" whose essence is captured at a glance. For closer inspection, it is easy to trace signals from component to component.

Of course, this cannot be claimed any longer for large, complex circuits. Here, formulas, i.e. text written in a formal notation, also

*Language*

*Formal substitution*

called a *language,* have their advantage, because they can be subjected to formal manipulation, i.e. to treatment according to the rules governing the notation. The most powerful tool in this context is *formal substitution.* Mathematical functions are specified with formal parameters, and the functions are applied to actual parameters. The meaning of such an application is obtained by the consistent substitution of every actual parameter for each occurrence of its corresponding formal parameter in the function's definition.

One may claim that also schematics can be structured, and they are. Circuits like multiplexers and counters are defined once only with their inputs and outputs labelled as formal parameters. Thereafter they appear in circuit descriptions in the traditional form of rectangular boxes, with the actual parameters connected to "pins". But as soon as structuring of this kind is used, the primary advantage of schematics is lost. It starts to decay into disconnected sub-pictures, and the tracing of signals along lines becomes impossible.

The advantage of easy parametrization of textual specification must not be underestimated. It allows formal texts to stand for large families of related circuits, such as e.g. for n-bit counters, where n is a formal parameter. In contrast, a picture always stands for itself only; it always represents a specific example.

Therefore - and not as often claimed because it is more easily read and processed by computers - the textual form is used more and more often. This does not imply that schematics are to be avoided at all cost. Small and perhaps intricate circuits may well be presented in this form, perhaps in addition to their textual specification. For large circuits, however, the textual form is inevitably preferable.

*Hardware description languages*

The text, however, must rest on a formal basis. Such formalisms are called *hardware description languages* (HDL). Two such "languages" with a growing popularity are VHDL and Verilog. They resemble procedural programming languages very strongly. A primary reason for this is their orientation toward circuit simulation, and hence such descriptions are regarded as executable programs, i.e. algorithmic processes. A genuine HDL text should, however, be a static description of a static object, namely the static circuit. No consideration of "dynamic behaviour" may enter the picture.

This does not prevent the adoption of many concepts from programming languages as fas as structure and syntax are concerned. With this in mind, we will present a small - almost minimal - formalism powerful enough to express synchronous circuits conveniently. Its name is *Lola*, short for Logic Language, short for logic circuit description language. Compilers for such languages may be used to automate circuit implementation at least partially. Systems indeed exist which compile a formal description into its ultimate realization, performing placement of components and routing of wires. Typically, such systems operate with the interactive support of the designer.

## 7.2. Lola: A Formal Notation for Synchronous Circuits

Our language's structure, its syntax, is defined in terms of the well-known Backus-Naur formalism BNF. We use a slightly extended form (EBNF) which allows us to express repetition and optionality in a convenient form. Arbitrary repetition of a sentential form s is expressed as {s}, and its optionality by [s].

A circuit description in Lola is called a *module*; compilers are supposed to accept modules as units of compilation. In the following description of Lola, the constituents of a module will be introduced, starting with the most elementary ones, and gradually building up more complex forms, until the entire structure of the module is defined.

*Module*

### 7.2.1. Identifiers, Integers, Logical Values, Reserved Words, and Comments

Identifiers are used to denote constants, variables, and types. They are sequences of letters and digits, starting with a letter, and optionally followed by an apostrophe. Typically, they are chosen by the programmer to reflect the meaning of the respective variable. In a formal sense, however, any identifier may be chosen, and a consistent substitution of all occurrences of an identifier by another one does not change the meaning of a text. The apostrophe is supposed to express the logical negation of the identifier's meaning. For example, if a variable is denoted by the identifier *reset'*, the intention is that the signal resets some circuit component if its value is 0 rather than 1. The apostrophe is not a negation operator of the circuit, but part of the identifier.

*Identifier*

The two possible values of a logic variable, true and false, are denoted by '1 and '0 respectively.

```
identifier    = letter {letter | digit} [" ' "].
integer       = digit {digit}.
LogicValue    = " '0" | " '1".
```

Certain sequences of capital letters are used as specific language symbols. They cannot be chosen as identifiers and are therefore called *reserved words.* They are listed below:

*Served words*

```
BIT      TS     OC
MUX      SR     LATCH  REG
DIV      MOD    CLOCK
BEGIN    IF     THEN   ELSE  ELSIF  FOR  DO   END
MODULE   TYPE   CONST  IN    INOUT  OUT  VAR  IMPORT
```

*Comments*

Comments are sequences of characters enclosed by the brackets (*and*). They may occur between any two symbols within a Lola text.

### 7.2.2. Basic and Simple Types

*BIT, TS, OC*

Every variable in Lola has a type. It is either a basic, predefined type or a structured, declared type. The basic types are denoted by BIT, TS, or OC, which differ only in their rules governing assignment. (TS denotes a tri-state bus, OC an open-collector bus). Variables of a basic type have a logic value, denoted by '0 or '1. In the case of a declared type, its declaration either is part of the module itself (see Sect. 7.2.10), or is provided in a module whose name is specified as prefix to the type identifier. The module name must be listed in the import list of the module's heading (see Sect. 7.2.9) and is said to be imported.

```
SimpleType     = BasicType | [identifier " . "] identifier ["("ExpressionList ")"].
BasicType      = "BIT" | "TS" | "OC".
ExpressionList = expression {" , " expression}.
```

### 7.2.3. Array Types

Array types consist of an array of elements, all of which have the same type. A numeric expression indicates the number of elements in the

array. Elements are identified by an index value. Indices range from 0 to the array's length minus 1. *Length*

```
type      = { "[" expression "]" } SimpleType.
```

### 7.2.4. Constant Declarations

Constant declarations serve to introduce identifiers denoting a constant, numeric value.

```
ConstDeclaration      = identifier ":=" expression ";".
```

### 7.2.5. Variable Declarations

Variable declarations serve to introduce identifiers denoting a variable and to associate it with a type. All variables declared in an identifier list have the same type.

```
VarDeclaration =      IdList ":" type ";".
IdList =  identifier {"," identifier}.
```

### 7.2.6. Expressions

Expressions serve to combine variables with logical operators to define new values. The operators are negation, logical conjunction (and), disjunction (or), and difference (xor). Operands are of any basic type. Elements of an array are selected by an index, for example *a.5*, *a.i.* If the index is an expression, the form *a[exp]* is used. *Operator*

- + logical disjunction (or)
- - logical difference (exclusive or)
- * logical conjunction (and)
- ~ negation (not)

A *multiplexer* is denoted by *MUX(s: a, b)* which is a short form for *~s*a + s*b*. Note that the selector parameter of MUX is followed by a colon instead of a comma. *Multiplexer*

A *register* provides the means to specify a value depending on previous values in time (sequential circuit). The value of *REG(e, d)* in the *next* clock cycle is equal to d in the current clock cycle, if e = '1. If e = '0, the previous value is retained. e is called the register's *enable* signal. The short notation *REG(d)* stands for *REG('1, d)*. *Register*

*Global clock*

In these forms, an implied, global clock signal is assumed, which is convenient in the case of synchronous circuits. To accommodate also the formulation of asynchronous circuits, a clock may be specified explicitly for each register by using the form *REG(ck: e, d)*. The abbreviation *REG(ck: d)* stands for *REG(ck: '1, d)*.

*D-latch*

*SR-latch*

A *D-latch*, denoted by *LATCH(e, d)*, is a storage element which holds a logic value while e = '0. If e = '1, the value d is acquired (i.e. the latch is transparent). An *SR-latch* with (active-low) set and reset inputs is expressed as SR(s', r').

*Numeric expressions*

Apart from logic expressions there exist numeric expressions. They follow the same rules of composition; their operators are those of addition, subtraction, multiplication, division, and exponentiation (↑). In the latter case, the base must be 2.

```
selector =   {"." identifier | "." integer | "[" expression "]"}.
factor =     ident selector | LogicValue | integer | "(" expression ")" |
             "~" factor |
             "MUX" "(" expression ":" expression "," expression ")" |
             "SR" "(" expression "," expression ")" |
             "LATCH" "(" expression "," expression ")" |
             "REG" "(" [expression ":"][expression ","] expression ")".
term =       factor {("*" | "/" | "DIV" | "MOD" | "↑") factor}.
expression = term {("+" | "-") term}.
```

### 7.2.7. Assignments

Assignments serve to define a variable's value, which is specified as that of an expression. The form v := x stands for "let v be equal to x". Hence, an assignment must be understood as a variable's *definition* (in contrast to an identifier's declaration). v and x do not have the same roles, and this asymmetry is emphasized by the use of the symbol := instead of the symmetric equal sign (=).

If a variable is of type BIT, the expression must be of any basic type, and only a single assignment (definition) is allowed.

If the variable's type is TS, the statement must specify a condition (representing a tri-state gate). Arbitrarily many assignments to the same variable are permitted. The (value of) the bus, however, is de-

fined only if some condition's value is '1.

```
assignment = identifier selector ":=" [condition "|"] expression.
condition  = expression.
```

If the variable's type is OC, arbitrarily many assignments to the same bus variable are permitted, too. The bus value is '1, unless any one of the assigned expressions has the value '0 ("*wired-or*").

### 7.2.8. Structured Statements

Statements are either assignments or composites of assignments, namely repeated or conditional assignments.

```
relation =          expression ("=" | "#" | "<" | "<=" | ">" | ">=")
                        expression.
IfStatement =       "IF" relation "THEN" StatementSequence
                        {"ELSIF" relation "THEN"
                        StatementSequence}
                    ["ELSE" StatementSequence]
                    "END".
ForStatement =      "FOR" identifier ":=" expression ".."
                        expression "DO" StatementSequence "END".
UnitAssignment =    identifier selector "(" ExpressionList ")".
statement =         [assignment | UnitAssignment | IfStatement |
                        ForStatement].
StatementSequence = statement {";" statement}.
```

The expressions in a for-statement must be numeric, and they specify the range of integer values for which the constituent statement sequence is defined. The identifier associated with the control variable is considered as being local to the for-statement, i.e. does not exist in the for-statement's context. The control variable typically serves as index to array variables.

### 7.2.9. Modules

A module specifies variables and a circuit involving these variables. A module may also contain definitions of composite types. Modules are the textual units for compilation.

```
InType =      {"[" expression "]"} "BIT".
InOutType =   {"[" expression "]"} ("TS" | "OC").
OutType =     {"[" expression "]"} ("BIT" | "TS" | "OC").
ImportList =  "IMPORT" identifier {"," identifier} ";".
module =      "MODULE" identifier ";" [ImportList]
              {TypeDeclaration ";"}
              ["CONST" {ConstDeclaration}]
              ["IN" {IdList ":" InType ";"}]
              ["INOUT" {IdList ":" InOutType ";"}]
              ["OUT" {IdList ":" OutType ";"}]
              ["VAR" {VarDeclaration}]
              ["CLOCK" expression ";"]
              ["BEGIN" StatementSequence]
              "END" identifier ".".
```

The import list specifies identifiers denoting modules, whose exported, composite types are referenced. Note that declarations introduce identifiers for variables, and statements define their values. The identifier at the end of the module's declaration must match the one following the symbol MODULE.

A clock definition serves to assign a specific expression to the implied, global clock, which is taken as default wherever no explicit clock parameter is indicated for a register.

Example: The following circuit represents an 8-bit binary adder with inputs x (x.0 ... x.7), y (y.0 ... y.7), and the carry ci. Its outputs are the sum s (s.0 ... s.7), and the carry co.

```
MODULE Adder;
  CONST N := 8;
  IN x, y: [N] BIT; ci: BIT;
  OUT s: [N] BIT; co: BIT;
  VAR c: [N] BIT;
```

```
BEGIN
   s.0 := x.0 - y.0 - ci;   c.0 := (x.0 * y.0) + (x.0 - y.0) * ci;
   FOR i := 1 .. N-1 DO
      s.i := x.i - y.i - c[i-1]; c.i := (x.i * y.i) + (x.i - y.i) * c[i-1]
   END ;
   co := c[N-1]
END Adder.
```

### 7.2.10. Composite Types and Unit Assignments

In addition to basic types and array types, composite types can be declared. This facility may be compared to record types in programming languages, and variables (instances) of such types correspond to components of circuits, i.e. to objects being part of a circuit. A type declaration specifies a composite type, of which instances are introduced by variable declarations. An asterisk following the type identifier indicates that the type may be referenced in other modules, i.e. that this type is exported.

The heading of a type declaration contains up to four sections:

1. The section headed by the symbol IN declares input signals to which no assignments within the type declaration are permitted. The identifiers act as formal names for expressions specified externally in unit assignments, where the expressions appear in the form of parameters. The types of the formal names must be BIT or arrays thereof. The corresponding actual expressions must be of any basic type, or an array thereof. *Input signals*
2. The section headed by the symbol INOUT declares bus signals to which assignments within the type declaration are permitted. As in the case of inputs, the identifiers act as formal names for signals declared outside the type declaration. Their types must be TS or OC or arrays thereof. *Bus signals*
3. The section headed by the symbol OUT declares actual variables. Their type must be BIT, TS, OC, or an array thereof. These output variables are accessible in the scope (type declaration) in which the composite variable is declared. There they are denoted by the composite variable's identifier followed by the output identifier as

selector (the latter acting like a field identifier of a record). No assignments are permitted outside the declaration in which the output is declared.

4. The section headed by the symbol VAR declares actual variables. They are not accessible outside the type declaration.

Summary:

| | allowed types | types of corresponding actual parameters |
|---|---|---|
| IN | BIT | BIT, TS, OC |
| INOUT | TS, OC | TS, OC |
| OUT | BIT, TS, OC | |
| VAR | BIT, TS, OC, declared type | |

Consider the following example:

```
TYPE AddElem;
   IN x, y, ci: BIT;
   OUT z, co: BIT;
   VAR h: BIT;
BEGIN h := x - y; z := h - ci; co := (x * y) + (h * ci)
END AddElem
```

A variable u of type *AddElem* (i.e. an instance of an *AddElem*) is introduced by the declaration:

```
u: AddElem
```

The inputs appear in the form of parameters (expressions) in a statement called *unit assignment:*

```
u(a, b, c)
```

The components of u are obtained by substitution of the actual expressions for the corresponding formal identifiers:

```
u.h := a - b;
u.z := u.h - c;
u.co := (a * b) + (u.h * c)
```

An 8-bit adder with inputs X and Y can now be declared as consisting of 8 identical elements

U: [8] AddElem

defined by the following assignments:

```
U.0(X.0, Y.0, '0);
FOR i := 1 .. 7 DO U.i(X.i, Y.i, U[i-1].co) END
```

and the sum is represented by the variables U.0.z ... U.7.z . (End of example)

```
TypeDeclaration = "TYPE" identifier ["*"] ["(" IdList ")"] ";"
                  ["CONST" {ConstDeclaration}]
                  ["IN" {IdList ":" InType ";"}]
                  ["INOUT" {IdList ":" InOutType ";"}]
                  ["OUT" {IdList ":" OutType ";"}]
                  ["VAR" {VarDeclaration}]
                  ["BEGIN" StatementSequence]
                  "END" identifier.
```

The number of expressions in a formal type specifies the number of indices used for this parameter. The expression indicates the length of the corresponding actual arrays specified in unit assignments. The identifier at the end of the declaration must match the one following the symbol TYPE.

### 7.2.11. Parametrized Types

Declared types can be supplied with parameters. They are numeric quantities and are used, for example, to parametrize the dimension of arrays. Example:

```
TYPE Counter(N);
   IN ci: BIT;
   OUT co: BIT; q: [N] BIT;
   VAR c: [N] BIT;
BEGIN q.0 := REG(q.0 - ci); c.0 := q.0 * ci;
   FOR i := 1 .. N-1 DO q.i := REG(q.i - c[i-1]); c.i := q.i * c[i-1] END;
      co := c[N-1]
END Counter
```

An instance u of a counter with 8 elements is declared as

u: Counter(8)

yielding the variables

u.co, u.q.0. ... , u.q.7 and u.c.0, ... , u.c.7

Note that u.c is local, i.e. not accessible outside the type declaration. A corresponding unit assignment of a counter with enable signal e is now expressed by u(e).

## 7.3. Examples of Textual Circuit Descriptions

Some of the circuits developed in the preceding chapters are specified below in terms of Lola component types. They include the adder (2.6), the multiplier (2.8), the universal shift register (4.2), the binary counter with clear signal (4.3), and the up/down counter (4.3). The first example, however, is a *barrel shifter*, a combinational circuit with $2^N$ inputs x.i, $2^N$ outputs z.i, and N inputs s.i. If s is considered as a binary coded number, then the output z is equal to x shifted (or rather rotated, hence the term barrel) by s positions, i.e. z.i : = x[(i+s) MOD N] for all i. The shifter is implemented by N stages of $2^N$ 2-to-1 multiplexers.

```
TYPE Barrel(N);
   IN s: [N] BIT;
   x: [2↑N] BIT;
   OUT z: [2↑N] BIT;
   VAR y: [N][2↑N] BIT;
BEGIN
   FOR j := 0 .. 2↑N-1 DO
      y.0.j := MUX(s.0: x.j, x[(j+1) MOD (2↑N)])
   END ;
   FOR i := 1 .. N-1 DO
      FOR j := 0 .. 2↑N-1 DO
         y.i.j := MUX(s.i: y[i-1][j], y[i-1][(j+2↑i) MOD (2↑N)])
      END
END ;
   FOR j := 0 .. 2↑N-1 DO z.j := y[N-1][j] END
END Barrel;
```

```
TYPE ASElement; (*add/subtract slice*)
   IN x, y, ci, s: BIT;
   OUT z, co: BIT;
   VAR u, h: BIT;
BEGIN u := y - s;  h := x - u; z := h - ci;  co := (x * u) + (h * ci)
END ASElement;

TYPE Adder(N);
   IN sub: BIT;
      x, y: [N] BIT;
   OUT co: BIT;
      z: [N] BIT;
   VAR AS: [N] ASElement;
BEGIN AS.0(x.0, y.0, sub, sub); z.0 := AS.0.z;
   FOR i := 1 .. N-1 DO AS.i(x.i,  y.i,  AS[i-1].co,  sub);  z.i := AS.i.z END;
   co := AS[N-1].co
END Adder;

TYPE AddElement;  (*add slice*)
   IN x, y, ci: BIT;
   OUT z, co: BIT;
BEGIN z := (x-y) - ci; co := (x * y) + ((x-y) * ci)
END AddElement;

TYPE Multiplier(N);
   IN x, y: [N] BIT;
   OUT z: [2*N] BIT;
   VAR M: [N][N] AddElement;
BEGIN
   FOR j := 0 .. N-1 DO M.0.j(x.0 * y.j, '0, '0) END ;
   FOR i := 1 .. N-1 DO
      M.i.0 (x.i * y.0, M[i-1].1.z, '0);
      FOR j := 1 .. N-2 DO  M.i.j(x.i * y.j, M[i-1][j+1].z, M[i][j-1].co) END
;
      M[i][N-1](x.i * y[N-1], M[i-1][N-1].co, M[i][N-2].co)
   END;
   FOR i := 0 .. N-2 DO z.i := M.i.0.z;  z[i+N] := M[N-1][i+1].z END;
   z[N-1] := M[N-1].0.z;  z[2*N-1] := M[N-1][N-1].co
```

```
   END Multiplier;

TYPE ShiftRegister(N);
   IN s0, s1: BIT; (*s1,s0 = 00: hold, 01: load, 10: up, 11: down*)
      x: [N] BIT;
   OUT y: [N] BIT;
BEGIN
   FOR i := 0 .. N-1 DO
      y.i := REG(MUX(s1, s0: y.i, x.i, y[(i-1) MOD N], y[(i+1) MOD N]))
   END
END ShiftRegister;

TYPE Counter(N);  (*with enable and clear*)
   IN en, clr': BIT;
   OUT Q: [N] BIT; co: BIT;
   VAR c: [N] BIT;
BEGIN Q.0 := REG((Q.0 - en) * clr'); c.0 := Q.0 * en;
   FOR i := 1 .. N-1 DO Q.i := REG((Q.i - c[i-1]) * clr'); c.i := Q.i * c[i-1]
   END ;
   co := c[N-1]
END Counter;

TYPE UpDownCounter(N);  (*with load, enable and clear*)
   IN ld', en, clr', up: BIT;
   OUT Q: [N] BIT;
   VAR cu, cd: [N] BIT;
BEGIN Q.0 := REG(MUX(ld': x.0, Q.0 - en) * clr');
   cu.0 := Q.0 * en; cd.0 := ~Q.0 * en;
   FOR i := 1 .. N-1 DO
      Q.i := REG(MUX(ld': x.i, Q.i - MUX(up: cd[i-1], cu[i-1])) * clr');
      cu.i := Q.i * cu[i-1]; cd.i := ~Q.i * cd[i-1]
   END
END UpDownCounter;
```

**Summary**

Hardware description languages have much in common with programming languages. Our small language Lola, for example, resembles syntactically the structure of the programming language Oberon. In HDLs, there also exist data types and instances thereof called variables. Composite types can be explicitly declared; their counterparts in programming are record structures. Whereas assignments to variables in programming reflect a dynamic process, a behaviour, their counterpart in HDLs is the definition of a variable's static value as a Boolean expression. Closely associated with the declaration of a composite circuit type is therefore the definition of its output variables' values. Consequently, the declaration of a circuit type combines the roles of record type and procedure declaration in programming languages.

Of particular relevance are the possibilities to construct arrays and matrices of variables, and to parametrize the declared circuit types. This gives rise to circuit libraries.

Computerized tools allow to process textual circuit specifications, i.e. to analyze them for various properties, to verify their consistency with implementations (layouts), and - ultimately perhaps - to generate implementations automatically.

# Design of an Elementary Computer

8.

**Overview**

The design techniques and circuit elements presented so far indeed suffice to build a complete, albeit simple computer. The prototype of the programmable, digital computer is the von Neumann architecture, which is both presented and implemented in this chapter.

The von Neumann architecture has remained the basic model of computers from its inception in 1945 until the present, and the number of fundamental concepts added in this long period has remained remarkably small. To mention are multiple instances of registers, the index register or, more generally, the computed address, the interrupt, and the use of several processors accessing a common store.

## 8.1. The Design of von Neumann

In the preceding chapters all the necessary components for designing a simple, digital computer have been introduced: combinational circuits, registers, busses, and memories. In this chapter we use them to build a concrete computer hardware. First, however, the question arises: Which property characterizes a circuit as a computer?

Since the times of the earliest automatic devices, in fact since Babbage's Analytical Engine around 1840, two units were distinguished in a computer: the *arithmetic unit* and the *control unit.* The arithmetic unit itself consists of two parts, the *function generator* capable of producing a sum (and possibly also logical sums and products) of two operands, and a *data store* (memory). The control unit also features two components: the *instruction sequencer* and the *instruction store* holding instructions in suitably encoded form. The sequencer - basically a state machine - in each step picks an instruction from the store, derives the control signals which determine the operation of the arithmetic unit, and computes the address of the location holding the next instruction, i.e. the instruction to be fetched and interpreted in the next step.

*Arithmetic unit*
*Control unit*

This scheme was fairly rigid in the sense that the instruction sequence was fixed after being loaded initially, and the device could then follow the steps of this single, invariant program only. What converted this rigid device into the modern, flexible computer were two fundamental ideas published in 1945 by *John von Neumann* (see Fig. 8.1):

1. The selection of the next instruction may depend on a result computed by the arithmetic unit.
2. Data store and instruction store are the same.

The first postulate has the far-reaching consequence that programs may contain conditional statements, and therefore also repetitive statements with computed termination conditions. The concept is easily implemented by a single signal leading from the arithmetic unit to the control unit, i.e. in the reverse direction of control signals determined by the current instruction.

The second postulate makes it feasible that a program may alter itself during execution, i.e. that instructions may be modified or replaced. This drastic measure is hardly used since the advent of higher-level programming languages, as it is not only a powerful tool but also a rich source of programming errors. The advantage of a unified data and instruction store rather lies in the fact that programs may be generated by other programs - we think of compilers - as data to be interpreted thereafter as instruction sequences. Furthermore, instruction addresses may be stored and later retrieved to serve as continuation points. We think of subroutines.

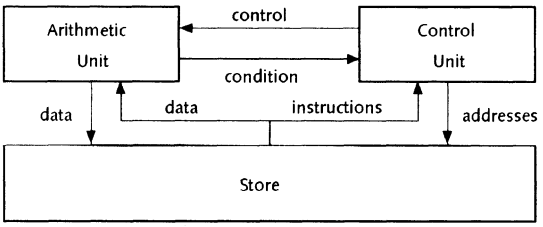



*Fig. 8.1. The structure of a computer according to von Neumann*

The basic structure of computers as postulated by von Neumann has remained remarkably stable during their subsequent evolution, which

otherwise has been characterized by extremely fast-changing technologies for implementation. The primary reason for this stability lies in its economy. Whenever a new technology emerged, it required a design of the least complexity possible in order to be applicable, and hence each time designers resorted to the scheme of von Neumann. It served as the conceptual basis in early vacuum tube computers (1950), and equally so for early types based on transistor technology (IBM 704, 1958). As transistor technology progressed and transistor prices fell, computer designs, so called „architectures", became more complicated, featuring not only faster units and larger stores, but also larger sets of more sophisticated instructions, including, for example, multiplication and division.

*Minicomputers*

The idea of combining modern transistor technology with the basic von Neumann structure resulted in the minicomputers of the early to mid-1960s (DEC PDP-1, PDP-8, HP 2116). In the following years, the trend to more sophisticated minis once more led to several embellishments of the basic scheme.

Then history repeated itself once more with the emergence of highly integrated circuit technology. In order to compress an entire processor into a single chip, it was once more only logical to return to the minimal design: the von Neumann scheme. Thus emerged the first microprocessors and microcomputers around 1975 (Intel 8080, Motorola 6800, and Rockwell 6502). Inevitably, it seems, designs became more complicated as soon as progress in miniaturization permitted, eventually leading to the CISC architectures (complex instruction set computers), epitomized by the Intel 80x86, Motorola 680x0, and National Semiconductor 32x32 processors of the mid-1980s.

*Microprocessors*

*CISC architectures*

*RISC architecture*

Now (1994) for the latest time, a return to the basics has occurred through the introduction of the RISC architecture (reduced instruction set computers) (IBM 801, MIPS, Sun SPARC, DEC Alpha). The subsequent, compelling trend towards more circuit complexity was this time not used so much to implement more complex instruction sets, as rather for gaining speed through instruction pipelining and caching.

It appears that history provides us with ample justification for considering the von Neumann scheme as fundamental and for studying it

in further detail, in fact, for using it as *the* application of the digital circuit fundamentals previously encountered.

Furthermore, the von Neumann scheme is directly reflected by those processors that are by far the most numerous in daily use: the *microcontrollers* (Intel 8051, Motorola 6805, Nat. Semi. COP 800). They are single-chip components containing not only the arithmetic unit and the control unit, but also a store and interfaces for input and output.

*Microcontrollers*

## 8.2. Choice of a Specific Architecture

In order to embark on the design of a simple processor, which we shall cynically call *Hercules,* we need to choose certain basic parameters, in particular the *instruction set* and the *instruction format.* According to the basic scheme, instructions consist of an *operation code* and a parameter, namely the *address* of the operand in store. The instruction set is partitioned into two categories, *data instructions* activating the arithmetic unit, and *control instructions* affecting the selection of the next instruction.

*Instruction format*

*Data instructions*

*Control instructions*

Other important parameters are the width of the data paths and the size of the store, which in turn defines the width of address lines. Although the minimal data path width is evidently 1, we choose for our design 8 with due regard for actually available circuit components. Hence the design is said to be *byte-oriented.* A benefit of this choice is that the store can be implemented with a single SRAM chip. Byte-wide SRAMs are available with capacities of $2^{15}$ bytes (256K bits) and more.

This choice in turn determines the instruction size. 15 bits are required for the operand address. About 5 bits are needed for a modest instruction set. Byte-orientation dictates that the instruction length be a multiple of 8, thus yielding 3 bytes for our example.

| | |
|---|---|
| Data path width | 8 |
| Address path width | 15 |
| Instruction format | op-code (1 byte), address (2 bytes) |

We now turn our attention to the choice of an instruction set. Among the data instructions, it is customary to include, besides addition, also the basic logical operations. Thereby the arithmetic unit becomes an

*arithmetic logic unit* (ALU). Furthermore, we add a single operation only, namely division by 2 (shift). *Arithmetic logic unit*

We notice that instructions specify a single operand only, whereas (binary) operators require two. The solution, also included in von Neumann's scheme, is to let one operand be implied, being the value held in a specific register (R). Because in early computers the addition of sequences of terms was a predominant operation, the sum was accumulated in this register, which is therefore traditionally called the *accumulator.* Its presence necessitates instructions for *loading* and *storing* in the accumulator R. *Accumulator*

The set of control instructions contains a variety of *branch* instructions. They allow one to divert the instruction stream to the point in the store specified by the branch instruction's parameter. The branch may be taken conditionally. The most obvious choices for conditions are the accumulator R holding zero (all bits 0), and holding a negative number (sign bit = 1). *Branch instructions*

Finally, we postulate a subroutine facility. It requires a branch instruction which deposits the address of the textually following instruction in a store in order that it may be retrieved for a return jump at the end of the subroutine. Since the subroutine branch instruction's parameter is already used to specify the branch destination, the place to hold the return address must be implied. We choose to provide a special, dedicated register (S) for this purpose. *Subroutine facility*

The preceding considerations lead to the following (provisional) instruction set, summarized in a mnemonic form: *Instruction set*

| | | |
|---|---|---|
| LOD | a | R := M[a] |
| STO | a | M[a] := R |
| ADD | a | R := R + M[a] |
| SUB | a | R := R - M[a] |
| AND | a | R := R AND M[a] |
| OR | a | R := R OR M[a] |
| XOR | a | R := R XOR M[a] |
| SHR | | R := R DIV 2 |
| BR | a | branch to a |
| BNE | a | branch to a, if R ≠0 (not equal to zero) |

| | | |
|---|---|---|
| BGE | a | branch to a, if R ≤ 0 (greater or equal to zero) |
| BGT | a | branch to a, if R > 0 (greater than zero) |
| BSR | a | S := adr of next instruction, branch to subroutine a. |
| RET | | branch to location given by S, return from subroutine. |

## 8.3. The Arithmetic Logic Unit (ALU)

As explained, the computational unit consists of a function generator, combining two operands to produce a result, and the implied accumulator register R. The former is a purely combinational circuit. The operands are the unit's input and R; the result is stored as the new value of R. Since all bits of a byte are subjected to the same operation, it is appropriate to compose the ALU of 8 identical components, called *slices*:

*ALU slice*

```
TYPE ALUslice;
  IN d, ci, si, zi: BIT;      (*data, carry-in, shift-in, zero-in*)
     seland, selor, selxor, seladd, zerox, invy, ldR, shR: BIT;
  OUT R, co, zo: BIT;         (*data, carry-out, zero-out*)
  VAR x, y, z, h: BIT;        (*operands, result of function unit*)
BEGIN
  x := ~zerox * R;
  y := d - invy;
  h := x - y;
  z := seland*(x * y) + selor*(x + y) + selxor*(x - y) + seladd*(h - ci);
  co := x*y + h*ci;
  R := REG(ldR, MUX(shR: z, si));
  zo := ~R * zi
END ALUslice
```

Given an array of 8 slices, they are joined into the ALU by the assignments

```
alu.0(d.0, ci, alu.1.R, '1, seland, selor, selxor, seladd, zerox,
  invy, ldR, shR);
FOR i := 1 .. 6 DO
   alu.i(d.i, alu[i-1].co, alu[i+1].R, alu[i-1].zo, seland, selor,
```

```
        selxor, seladd, zerox, invy, ldR, shR)
END ;
alu.7(d.7, alu.6.co, C, alu.6.zo, seland, selor, selxor, seladd, zerox,
   invy, ldR, shR)
```

The complete ALU is schown schematically in Fig. 8.2, and it requires some additional clarifications which follow.

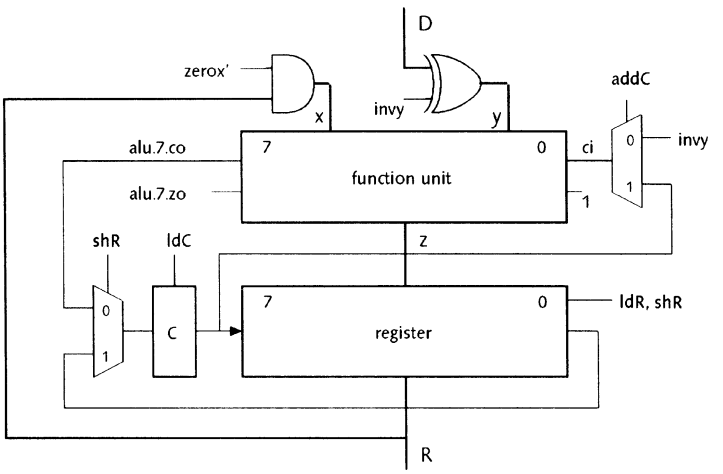



*Fig. 8.2. ALU schema*

**Comments**

1. The inclusion of *and* gates in the path of the left operand x permits the direct loading of R with the data input D when zerox' = 0 with function = *or, xor.*
2. The inclusion of *xor* gates in the path of the right operand y permits the complement of D to be taken as y. This facility is used for subtraction, and is also useful for logical operations with complemented second operand.
3. In order to be able to record an overflow (carry out), an additional (single-bit) register C is provided, which together with R holds the 9-bit sum. *Register C*
4. For adding (subtracting) multiple-byte operands through consecutive operations, the carry output held in register C must be

accessible as carry input for the next byte addition. This is achieved by placing a multiplexer at the ALU's carry input. The multiplexer is controlled by the *addC* signal which selects C for the additional ADDC (add with carry) instruction.

ci := MUX(addC: invy, C)

5. The shift path used in the right shift instruction includes register C, and is circular. The shift therefore involves a rotation path of length 9. This is necessary for implementing shifts over multiple-byte operands. C then serves for holding the bit to be shifted over into the next byte. The corresponding instruction will be named ROR (rotate right) instead of SHR (shift right).

   C := REG(ldC, MUX(shR: alu.7.co, alu.0.R))

*ALU signals*

In summary, the ALU's interface consists of the following signals:

| | | |
|---|---|---|
| Inputs: | D.0 ... D.7 | data |
| Outputs: | R.0 ... R.7, C, zo | |
| Controls: | seland, selor, selxor, seladd | (function selection) |
| | zerox' | (force x to 0) |
| | invy | (invert D input) |
| | addC | (add carry C) |
| | ldR, ldC, shR | (load R, load C, shift R) |

## 8.4. The Control Unit

The principal functions of the control unit are to hold the instruction currently being interpreted, thus permitting the derivation of control signals which determine the operation of the ALU (and of the control unit itself), and the computation of the address of the next instruction.

*Instruction register*

It follows immediately that the control unit must contain an *instruction register* consisting of a part (IR) holding the operation code and a

*Address register*

part holding the parameter, the *address register* (A). Also a register holding the address of the next instruction is indispensible. It is com-

*Program counter*

monly called the *program counter* (PC). A further register (S) serves to hold a subroutine return address.

Because the selection of the next instruction may be conditional, a multiplexer is required at its input, selecting either PC (no branch)

or A (branch). PC is incremented after each instruction fetch.
These considerations result in a control unit structure shown schematically in Fig. 8.3. The part generating the next address is called a *sequencer.*

*Sequencer*

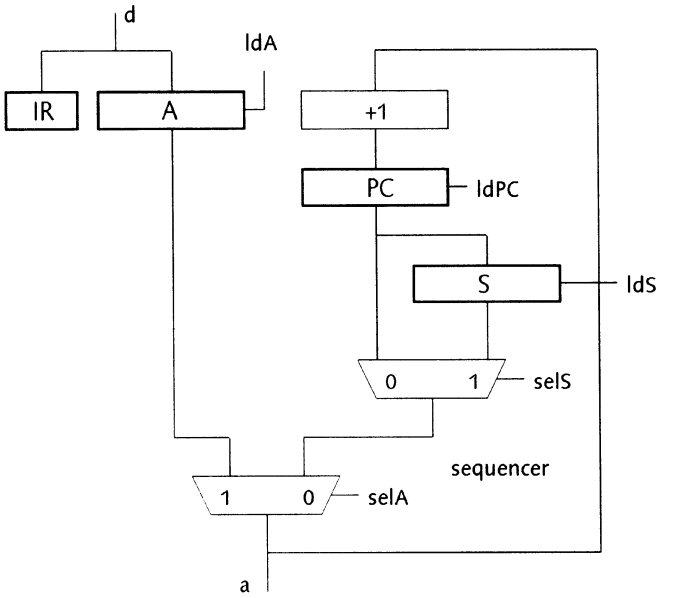



*Fig. 8.3. The structure of the control unit*

As in the case of the ALU, the sequencer is conveniently partitioned into slices, whose number is determined by the address width. A slice is defined as follows:

```
TYPE AdrGenSlice;
   IN d, ci: BIT;
      selA, selS, ldPC, ldS, ldA: BIT;  (*controls*)
   OUT a, co: BIT;
   VAR PC: BIT;        (*program counter*)
      A: BIT;          (*address register*)
      S: BIT;          (*return address register*)
BEGIN
   PC := REG(ldPC, a - ci);  co := a * ci;  (*incrementer*)
   A := REG(ldA, d);  (*address register*)
   S := REG(ldS, PC);
```

```
    a := MUX(selA: MUX(selS: PC, S), A)
END AdrGenSlice
```

The complete control unit consists of the parts declared as

```
IR:   [8] BIT;
ag:   [15] AdrGenSlice
```

and defined by

```
FOR i := 0 .. 7 DO IR.i := REG(ldIR, d.i) END ;
ag.0 (d.0, '1, selA, selS, ldPC, ldS, ldA0);
FOR i := 1 .. 7 DO
   ag.i (d.i, ag[i-1].co, selA, selS, ldPC, ldS, ldA0)
END;
FOR i := 8 .. 14 DO
   ag.i (d[i-8], ag[i-1].co, selA, selS, ldPC, ldS, ldA1)
END
```

Summarizing, the control unit has the following interface:

| | | |
|---|---|---|
| Inputs: | d.0 ... d.7 | input from store |
| | ldIR, ldA0, ldA1, ldPC, ldS, selA, selS | controls |
| Outputs: | a.0 ... a.14 | next instruction address |
| | IR.0 ... IR.7 | current opcode |

## 8.5. Phase Control and Instruction Decoding

A fundamental postulate of the von Neumann scheme is that processing of each instruction proceeds through phases. The first phase is the *fetch phase*, in which an instruction is fetched from the store and loaded into the instruction register (IR, A). The second phase is the *execution phase*, in which the specified operation is performed according to the control signals derived from the value held in the instruction register.

*Fetch phase*

*Execution phase*

In the case of our example, every instruction consists of 3 bytes. Since at any time only a single byte can be read from the store, the fetch phase must be further subdivided into 3 cycles. Hence, every instruction cycle consists of 4 subcycles in total, the last of which is the execution phase. It follows that the machine's control state is deter-

*Subcycles*

mined by phase signals in addition to the derived control signals. For the generation of the phase signals, the obvious choice is a state machine. And because only 4 phases occur, we choose a one-hot solution:

*Phase signals*

```
PH: [4] BIT;  (*initialized to 1, 0, 0, 0*)
FOR i := 0 .. 3 DO PH.i := REG(PH[(i-1) MOD 4]) END
```

From the specified roles of the phases we obtain:

```
ldIR := PH.0;
ldA0 := PH.1;
ldA1 := PH.2;
ldPC := ~PH.3;  (*PC incremented except in phase 3*)
```

All other control signals depend on IR, because they depend on the instruction to be interpreted. The choice of instruction encoding should be such that the expressions for the control signals become reasonably simple. Without detailed explanations, let us fix the following encodings. We merely point out that the class of an instruction (data or control) is determined by a single bit (IR.3).

| IR | 7 | 6 | 5 | 4 | 3 |
|---|---|---|---|---|---|
| ROR | 0 | 0 | 0 | 0 | 1 |
| ADD | 0 | 1 | 0 | 0 | 1 |
| ADDC | 0 | 1 | 0 | 1 | 1 |
| LOD | 1 | 0 | 0 | 0 | 1 |
| XOR | 1 | 0 | 0 | 1 | 1 |
| OR | 1 | 0 | 1 | 0 | 1 |
| AND | 1 | 0 | 1 | 1 | 1 |
| STO | 1 | 1 | 1 | 1 | 1 |

This yields expressions for the ALU controls:

```
seland :=  IR.7 * ~IR.6 * IR.5 * IR.4 * IR.3;        AND
selor :=   IR.7 * ~IR.6 * IR.5 * ~IR.4 * IR.3;       OR
selxor :=  IR.7 * ~IR.6 * ~IR.5 * IR.3;              XOR, LOD
seladd := ~IR.7 * ~IR.5 * IR.3 * PH.3;               ADD, ADDC, ROR
addC :=   ~IR.7 * IR.6 * ~IR.5 * IR.4;               ADDC
```

```
zerox :=   IR.7 * ~IR.6 * ~IR.5 * ~IR.4;                LOD
shR :=   ~IR.7 * ~IR.6 * ~IR.5;                         ROR
ldR :=   ~(IR.7 * IR.6 * IR.5) * IR.3 * PH.3;           all except STO
sto :=     IR.7 * IR.6 * IR.5 * IR.4 * IR.3 * PH.3;     STO
```

Furthermore, we let

```
invy := IR.2
ldC := seladd
```

For control instructions we choose the following encodings:

| IR | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | branch always |
| BNE | 0 | 0 | 0 | 1 | 0 | 0 | 0 | | branch, if not zero |
| BGE | 0 | 0 | 0 | 0 | 0 | 1 | 0 | | branch, if greateror equal to zero |
| BGT | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | branch, if greater than zero |
| BCC | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | branch, if carry clear (C = 0) |
| BSR | 0 | 0 | 0 | 0 | 0 | 0 | 1 | | branch to subroutine |

By simply defining

```
selS := IR.0 * ~PH.0
```

*Subroutine return*

we let all instructions with IR.0 = 1 additionally act as a subroutine return, thus making a specific subroutine return instruction superfluous. And finally we define the address output control *selA* to select the address register in the following two cases: (1) during phase 3 if a data instruction is interpreted, and (2) during phase 0 if the instruction is a branch and if the branch is not inhibited.

```
selA := MUX(IR.3: ~PH.0 * cond, PH.3)     branch, if branch
                                          instruction and cond
cond := ~(IR.2 * alu.7.R  +  IR.4 * alu.7.zo  +  IR.5 * C)
                                          branch condition
```

And this completes the specification of our Hercules computer. Almost. As it stands, it serves well to demonstrate the principle of the von Neumann computer and to derive an implementation with digital circuit components. For practical use, however, it is incomplete, because it neither accepts inputs nor generates any outputs. There is

no way to load its program, either. The latter problem is briefly addressed at the end of Sect. 8.6, and a suggestion for including some simple input and output facilities will be presented in Chap. 11.

## 8.6. An Implementation Using Standard Parts

To conclude this chapter, we proceed to build our Hercules computer using commercially available, standard parts, integrated circuit components of the TTL family. We shall arrive at a chip count of some two dozen units, thus making it possible to assemble the entire circuit on a small, experimental board using the wire-wrap technique. The result is shown schematically in Fig. 8.4 and Fig. 8.5. For further details about the chosen components the reader is referred to their respective data books.

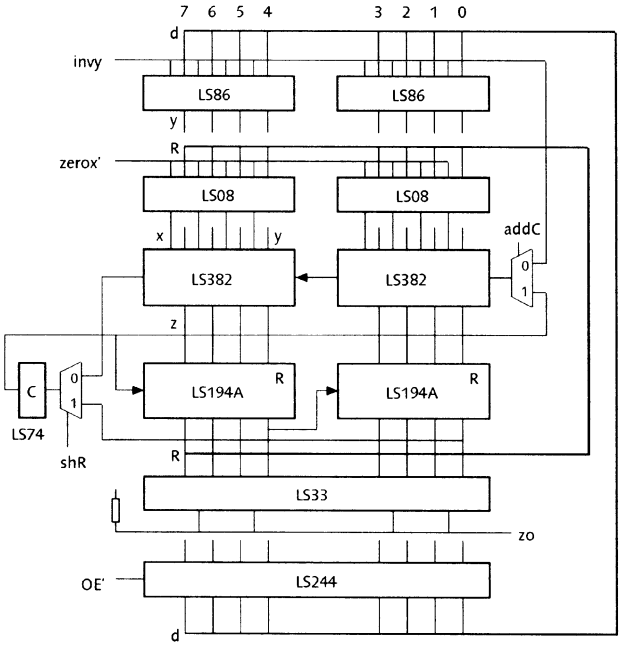



*Fig. 8.4. Arithmetic Unit*

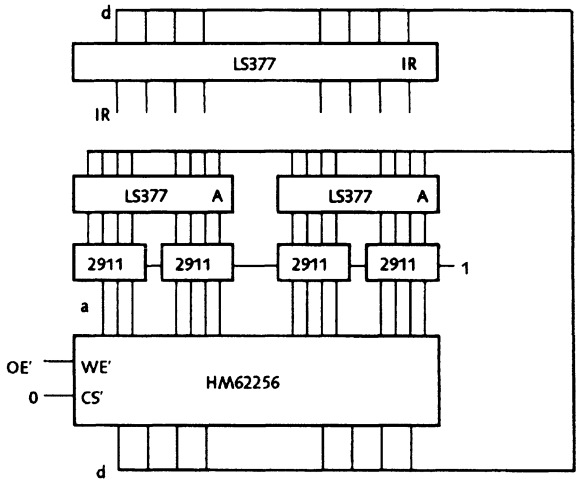



*Fig. 8.5. Control Unit*

*Summary of parts:*

| | |
|---|---|
| LS08 | quadruple 2-input And-gates |
| LS33 | quadruple 2-input Nor-gates with open collectors |
| LS74 | two D-registers |
| LS86 | quadruple 2-input Xor-gates |
| LS194A | 4-bit shift register |
| LS382 | 4-bit function generator |
| LS244 | 8 tri-state drivers |
| LS377 | 8-bit D-register with enable |
| Am2911 | 4-bit sequencer slice |
| HM62256 | 32K x 8 SRAM |

The circuit presented differs slightly from the one previously developed. It does so because components are available which almost satisfy specific needs, but deviate from the given specifications in some detail. If an accommodation is easily possible, it may contribute to a reduction of the chip count. The choices of parts and possible differences to the previous model are explained as follows:

1. For the ALU, the function generator component LS382 is chosen. It is a cascadable 4-bit slice. Its function is selected by the three control inputs S0, S1, and S2.

| S2 | S1 | S0 | function F |
|---|---|---|---|
| 0 | 1 | 1 | A plus B |
| 1 | 0 | 0 | A xor B |
| 1 | 0 | 1 | A or B |
| 1 | 1 | 0 | A and B |

2. The accumulator register is represented by two 4-bit shift registers LS194A. Its function is controlled by the signals S0, S1.

| S1 | S0 | function | |
|---|---|---|---|
| 0 | 0 | hold | |
| 0 | 1 | shift left | left input = LI |
| 1 | 0 | shift right | right input = RI |
| 1 | 1 | load | from D |

3. The condition signal zo (all R.i = 0) is generated by an LS33 chip containing four 2-input *nor* gates with their open-collector outputs tied together.
4. The store is implemented by a single HM62256 static RAM chip. It uses the same pins for data input and output, thus forcing the data signals to be a bus. The chip includes tri-state drivers controlled by signal OE' (output enable, active low). Signal WE' specifies a write operation when low. Since there is only one RAM chip involved, its selection signal CS' is always held active (low).
5. Because the data lines function as a bus, the ALU output needs to be connected via tri-state gates, too. A single LS244 chip serves for this purpose, containing 8 tri-state drivers. G0 and G1 are the enable inputs.
6. The 3-byte instruction register IR, A is represented by three D-type register chips LS377.
7. With the exception of the instruction register, the entire control unit is implementable by four cascadable 4-bit sequencer slices Am2911 (Fig. 8.6). They contain the PC register, the S register holding a return address, and the multiplexer for selecting the next address. In fact, the S register is implemented as a stack capable of holding up to 5 return addresses. The signals controlling the function of the Am2911 are called S1, S0, FE', PUP, OE', and ZERO'. OE'

is held low. ZERO' forces all address outputs to 0, if low; it is therefore connected with the global reset signal to start instruction fetch from location 0.

| S1 | S0 | address selection | | FE' | PUP | stack control | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | PC | continue | 0 | 0 | pop S | return |
| 1 | 0 | S | return | 0 | 1 | push S | branch subroutine |
| 1 | 1 | D | branch | 1 | - | no change | |

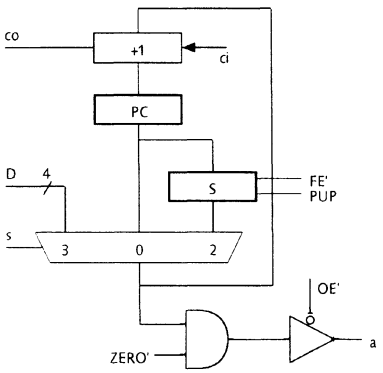



*Fig. 8.6. The Am2911 sequencer element*

*Phase signals*

8. The state machine generating the four phase signals is implemented as a one-hot circuit using an LS194A 4-bit shift register. S1 is held high, and S0 is connected to the inverted global reset signal, causing the register to be loaded when active (low). Input D0 is 1, D1 = D2 = D3 = 0. Hence, after a reset, phase 0 is indicated.
9. The obvious choice for decoding the instruction code and generating the control signals are combinational PLDs. Two instances of the type PAL16L8 suffice. Note that the set of controls is not identical to the one previously introduced, although they have the same purpose. This is due to the fact that the control input specifications of LS382, LS194A, and Am2911 must be respected.
10. The 2-to-1 multiplexers for the carry input of the function generator and for the input of the C register are implemented by two 4-to-1 multiplexers contained in a single LS153 component. This turns out to lead to a more complex circuit, but yields a minimal chip

count (1). Note that the multiplexers share the selector signals (see Fig. 8.7).

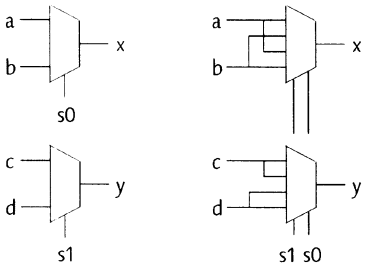



*Fig. 8.7. Using two parallel 4-to-1 multiplexers instead of two independent 2-to-1 multiplexers*

11. The heart of the entire computer, the center that makes the thing tick, is the generator of the global clock signal. The easiest solution is to use an integrated crystal oscillator (up to 4 MHz). But also a simple RC oscillator will suffice for this experimental setup, if a frequency of, say, 100 kHz is considered adequate. An oscillator circuit based on the popular NE555 component is shown in Fig. 8.8. *Clock signal*
12. The global reset signal RST' originates from a push-button. In order to obtain a clean, single transition, it must be „debounced". Although a simple SR latch would suffice, the use of the TL7705 component is recommended to generate a signal of prescribed duration (see Fig. 8.8). *Reset signal*
13. In order to make register states visible, light emitting diodes (LEDs) may be used. They are to be „driven" by appropriate circuitry, for example as shown in Fig. 8.8 by inverters with open collector outputs. The unit 7406 contains 6 such elements. A current-limiting resistor of 270 ohm yields a current of about 6 mA when the inverter input is 1. *LED*

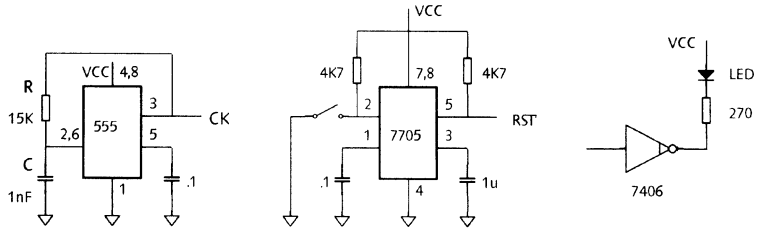



*Fig. 8.8. Circuits to generate the clock and reset signals, and to connect light emitting diodes*

14. Typically, computers feature a read-only store in addition to the writable store. The purpose of the ROM is to hold a small program which serves (primarily) to load programs into the RAM. Even if Hercules does not so far provide any facility for input - such an addition will be discussed in Chapter 11 - the addition of a ROM is indispensible to hold a simple, fixed test program. A ROM can easily be added due to the bus nature of the data path between ALU and store, and it is shown in Fig. 8.9. The ROM is enabled, if the address lines a.k ... a.14 have value 0 (addresses 0 ... 2k-1), and the RAM will be active otherwise.

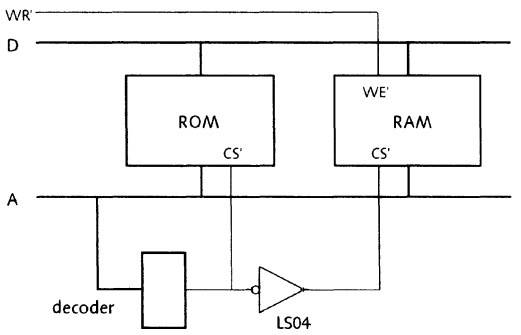



*Fig. 8.9. Addition of a read-only store*

## 8.7. Interrupts

Considering the phenomenal advances of computer technology in the last decades, it is rather surprising how few fundamental concepts had been added to the scheme of von Neumann since 1945. Apart from facilities to compute addresses (addressing modes), it is primarily the interrupt facility which both is fundamental and has added considerable flexibility to the computer.

The most obvious deficiency of the computer architecture presented so far is the lack of a connection to external signals influencing program control. Although we defer their introduction to Chapter 11, it is easy to realize that conditional branch instructions could be made to depend not only on internal signals such as the sign bit of the accumulator, but also on external signals. The problem with this facility is that the influence of such signals would be restricted to moments when program interpretation reaches a branch instruction testing an

external signal. At all other times, the computer must be considered busy and deaf. If a program is to wait until an external signal becomes active, repeated testing in a tight loop is used. This technique is called *polling*. Naturally, the computer is then unavailable for other tasks, and polling is therefore a kind of busy waiting.

*Polling*

In certain applications it is mandatory that the computer reacts without delay to an external signal. By „reaction" is meant its diversion to a specific instruction sequence expressing the computer's reaction to the signal. This diversion is most easily implemented as a branch instruction. It implies that the current instruction sequence is interrupted and, obviously, should be resumed after the request had been serviced. The branch must therefore be implemented like a subroutine branch, and the subroutine is called the *interrupt handler*. The external request signal is an *interrupt signal*.

*Interrupt signal*

How, then, is a branch instruction (standing for the entire interrupt handler) inserted into the current instruction sequence? Effectively, the interrupt signal must be polled after *every* instruction executed. This is implicitly and without delay achieved by expanding the state machine which determines the interpretation cycle and generates the phase signals. A special interrupt phase (PH.4) is added. Its flow diagram and circuit are shown in Fig. 8.10. The execution of the implied branch (the interrupt response) takes only a single cycle, because the instruction need not be fetched from the store.

*Interrupt phase*

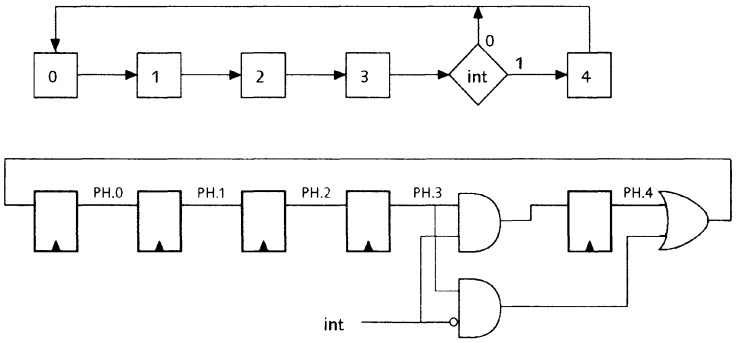



*Fig. 8.10. Flow diagram and circuit of extended state machine*

The new definitions of the state signals are:

```
PH.0 := REG(PH.3 * ~int + PH.4);
PH.1 := REG(PH.0);
PH.2 := REG(PH.1);
PH.3 := REG(PH.2);
PH.4 := REG(PH.3 * int)
```

It follows that an interrupt is serviced only after the completion of the interpretation cycle of the current instruction, because *int* is sampled only after phase 3. Also, the first instruction of the interrupt handler is interpreted unconditionally.

There remains the question of what should be the destination address of the implicit branch, which is the location of the interrupt handler. It is customary to fix this address in the design as a distinguished value. Since 0 is already taken as the starting location after reset, we choose the value represented by all ones for reasons of simplicity. Then, only the following signal definitions need be changed:

```
a := MUX(selA: MUX(selS: PC, S), A) + PH.4;    (in AdrGenSlice)
selS := IR.0 * ~PH.0 + PH.4
```

*Interrupt enable*

It is indispensible that a computer's sensitivity to interrupts may be suspended under program control. This calls for a state variable *ien* (interrupt enabled), i.e. for a flipflop, whose value determines whether or not the interrupt signal may take effect (int := extint * ien). This flipflop is set under program control (special instruction) and reset by PH.4, which is also under program control. In most computers, this *ien* bit is contained in a so-called *program status register.*

*Program status register*

The implementation of the interrupt facility is surprisingly simple. It must be noted, however, that in the solution shown here, the interrupt signal is assumed to be synchronous with the computer's clock.

**Summary**

The von Neumann architecture partitions a computer into three parts: the arithmetic-logical unit (ALU), the control unit, and the common store for instructions and data. The ALU consists of a purely combinational function generator and of a register, traditionally called the accumulator. Typical functions are addition and the basic Boolean operators. An additional single-bit register serves to hold the carry output of addition. The arguments of the available functions are data held in the accumulator and in memory.

The control unit serves to address consecutive instructions in store, and to fetch them into the instruction register (IR) for decoding and interpretation. For this purpose, the control unit contains a register called the program counter (PC), whose value is incremented after each instruction cycle. A sequence can be broken by branch instructions, which specify the address of the next instruction explicitly.

An interrupt represents the break of an instruction sequence not by a programmed branch, but instead by the occurrence of an external event.

# Multiplication and Division

9.

**Overview**

Multiplication and division are of a higher order of complexity than addition and subtraction. They are therefore not among the ALU-operations of the presented elementary computer. Instead, we explain efficient algorithms for multiplying and dividing by repeated addition and subtraction. Slight extensions of the basic ALU hardware allow for substantial acceleration of these programs. We thereby demonstrate the close interrelationship between algorithm and architecture.

Multiplication and division can be composed as sequences of additions and subtractions. They are therefore not included as instructions in minimal computer architectures. In this chapter we present two fundamental, efficient algorithms for multiplying and dividing natural numbers (unsigned integers). As shown in Chapter 2, multiplication using a combinational circuit to obtain the product immediately requires a substantial amount of circuitry, if the number of digits lies in a useful range. The solution for circumventing this expense is to compute the product in sequential, identical steps. The same method is used for division.

The first question arising is: What is to be done in such a step? After answering this question, we will program it in terms of Hercules instructions, thereby demonstrating that reasonably fast multiplication is possible even with a computer of such primitive structure. The same demonstration will be given for division.

The next question to be answered is: Could the primitive computer be augmented with moderate effort to let the step be expressed by a single instruction? The somewhat surprising result is that this is indeed possible, thus speeding up multiplication and division very considerably. The two examples serve to demonstrate the substantial benefits of close cooperation between hardware and software design.

## 9.1. Multiplication of Natural Numbers

The simplest way to calculate the product z = XY in a sequence of steps is to add Y (the multiplicand) to a partial sum X times. The algorithm will consist of a sequence of identical steps. To develop the algorithm and to convince ourselves of its correctness, we postulate an *invariant* expression (predicate). This invariant must be true before and after each step, and therefore also at the beginning and end of the algorithm. It must be chosen so that together with the termination condition of the repetition of steps, it yields the desired result.

Using variables x, y, and z - the latter for the partial sum - an appropriate invariant is

P: xy + z = XY

where X and Y are the multiplier and multiplicand. The initial assignments

x := X; y := Y; z := 0

make P trivially true. A possible invariant preserving step which brings the computation closer to its goal x = 0 is

x := x-1; z := z+y

The algorithm then is

```
x := X; y := Y; z := 0;
WHILE x # 0 DO x := x-1; z := z+y END
```

and the result follows from P and x=0, i.e. z = XY.

Unfortunately, this algorithm is not only simple, but also inefficient, since the number of required steps is X, which may be large. We therefore have to find a step that decreases x more rapidly, i.e. by larger amounts. An obvious idea is to halve x instead of subtracting 1 only. The pair of operations

x := x DIV 2; y := 2*y

indeed preserves the invariant P, but only if x is initially even. We therefore let this pair be preceded by another invariant preserving pair, namely the one used in the first solution above:

x := x-1; z := z+y

and obtain an efficient algorithm based on the same invariant P. We point out that doubling and halving can be implemented by a simple shift operation.

```
x := X; y := Y; z := 0;
WHILE x # 0 DO
   IF ODD(x) THEN x := x-1; z := z+y END ;
   x := x DIV 2;  y := 2*y
END
```

The following, third solution is even better suited for implementation on a simple computer. It is merely a variant of the algorithm presented so far, and relies on the fact that numbers are represented by a fixed number N of bits. It therefore suffices to perform exactly N steps, as halving the largest representable multiplier N times reduces it to 0. An advantage of this solution, as will become apparent when coding it in terms of machine instructions, is that both x and z are being divided, i.e. shifted in the same direction. We introduce a further variable i, counting the number of steps, and use the slightly modified invariant

$$P': \quad (2^N xy + z = 2^i XY) \;\&\; (0 <= x < 2^i)$$

and the following triple of assignments, which notably preserves P':

```
x := x DIV 2; z := z DIV 2; i := i - 1
```

The new algorithm then assumes the following form consisting of n *add-shift steps*

```
x := X; z := 0; i := N;
REPEAT (* P' *)
   IF ODD(x) THEN z := z + 2^N y; x := x - 1 END
   z := z DIV 2; x := x DIV 2; i := i - 1
UNTIL i = 0
```

*Add-Shift step*

The repeated statements are called the *Add-Shift step.* The verification of the desired result is obtained by the following implications: From $i = 0$ and $0 <= x < 2^i$ (second part of P'), we conclude $x = 0$, from which, together with the first part of P', we conclude $z = XY$.

### Comments

1. The statement x := x-1 turns out to be superfluous, since the subsequent integer division of x discards the remainder anyway.
2. Addition of $2^N y$ is achieved by adding y shifted by the *constant* amount of N digits.
3. The predicate ODD(x) is determined by the least significant bit of x.

Since the product of two 2N-bit factors requires $2^N$ bits for its correct representation, z evidently requires a register of double length. It is possible, however, to implement this algorithm using, in addition to the double register for z, only one additional register for y. The solution lies in sharing the double register for both z and x. In fact, the double length property manifests itself only in the requirement that two single-length registers may be considered as joined with respect to shifting. This scheme is illustrated by Fig. 9.1, where a left part of the double register is shown to be occupied by the partial sum z, and its right part by the multiplier x. Register M is called the *multiplier register.*

*Multiplier register*

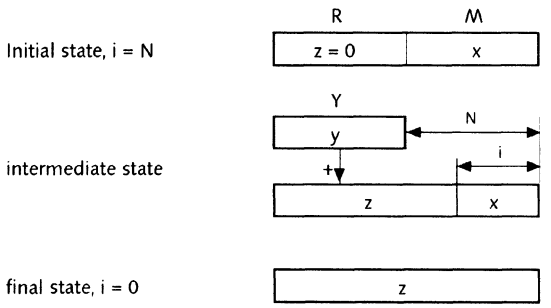



*Fig. 9.1. Multiplication steps using registers R, M and Y*

The process of multiplying y = 13 with x = 11, based on 4-bit registers (N = 4) is shown as an explanatory example (x in italics). Note that Y is held constant:

| Y | 1101 | | y = 13 |
|---|---|---|---|
| | R | M | |
| z, x = | 0000 | *1011* | z = 0, *x = 11* |
| +y | 01101 | *1011* | step 3 |
| → | 0110 | 1*101* | |
| +y | 10011 | 1*101* | step 2 |
| → | 1001 | 11*10* | |
| +0 | 01001 | 11*10* | step 1 |
| → | 0100 | 111*1* | |
| +y | 10001 | 111*1* | step 0 |
| → | 1000 | 1111 | z = 143 |

We now proceed to code the presented add-shift step in terms of Hercules instructions, and obtain:

| Step | LOD | M | Multiplier x |
|---|---|---|---|
| | BEV | L1 | branch if x even |
| | LOD | R | z |
| | ADD | Y | |
| | STO | R | z := z + Y |
| L1 | LOD | R | |
| | ROR | | |
| | STO | R | z := z DIV 2 |
| | LOD | M | |
| | ROR | | |
| | STO | M | x := x DIV 2 |

If this sequence is concatenated N times, the counter variable i emerges as a dummy variable and can be omitted. We observe that we have actually cheated, in the sense that we use a conditional instruction that does not actually occur in the instruction set of Hercules: BEV. However, we also note that a test of the least significant bit of the accumulator can be achieved by rotating, thereby bringing the bit into the C register and then using the BCC instruction. The reader should convince himself that the following, *shorter* variant is correct, if the N steps are followed by the additional three instructions labelled with L2.

```
Step    LOD    M        Multiplier x
        ROR             x := x DIV 2, remainder in C
        STO    M
        LOD    R        z
        BCC    L1
        ADD    Y        z := z + Y
L1      ROR             z := z DIV 2
        STO    R

L2      LOD    M
        ROR
        STO    M
```

## 9.2. Division of Natural Numbers

Just as multiplication can be achieved by repeated addition, division is achieved by repeated subtraction. Let

X = dividend
Y = divisor
q = quotient
r = remainder

Then integer division is defined by the relations

$$X = qY + r, \quad 0 \leq r < Y$$

The division algorithm evidently consists in starting with r = X and in subtracting Y from r as long as $r \geq 0$, while always maintaining the invariant suggested by the definition of division:

$$P: (X = qY + r) \,\&\, (r \geq 0)$$

The invariant is trivially established by the initial assignments q := 0 and r := X, and preserved by the pair of assignments

```
r := r - Y; q := q+1
```

Thus, the algorithm is given by the simple program

```
r := X; q := 0;
WHILE r ≥ Y DO r := r - Y; q := q+1 END
```

and the invariant P together with the termination condition r < Y establishes the postulated result.

Unfortunately, once again this solution is not only simple, but also inefficient. The key to a better solution lies in decrementing r in larger steps. As in the well-known method for „division by hand", we choose the subtrahend as Y multiplied by a power of the number system's base, here 2, and start out with a power as large as possible.

As in the case of multiplication, we use the fact that all numbers are represented by a fixed number of digits (N), and therefore start by trying to subtract Y multiplied by $2^N$ from the remainder r. We will subsequently use the identifier Y' to denote $2^N Y$. Instead of subsequently halving the subtrahend Y' in each step, r is doubled; the advantage is that both r and q are doubled, i.e. shifted in the *same* direction. This suggests that as in the case of multiplication, r and q can share a double register, since r is steadily decreasing, whereas q is increasing. The resulting algorithm repeating the same *shift-subtract step* N times is formulated as shown below. The (auxiliary) variable i is the step number, and instead of P we use a slightly adapted invariant

*Shift-subtract step*

$$P': (qY' + r = 2^i X) \ \& \ (0 \leq r < Y')$$

which is trivially established by the initial assignments. Note that in order that both q and r can be represented by N bits ($q, r < 2^N$), the initial condition X < Y' must be postulated (and furthermore $X < 2^{2N-1}$).

```
r := X; q := 0; i := 0;
REPEAT  (* P' *)
   r := 2*r; q := 2*q; i := i+1;
   IF r ≥ Y' THEN r := r - Y'; q := q+1 END
UNTIL i = N
(* (qY' + r = 2^N X) & (0 ≤ r < Y') *)
```

As in the case of multiplication, a double-length register is initially used for the remainder, and it is shared by the (shrinking) remainder r and the (growing) quotient q, thus saving resources. Register Q is called the quotient register. This scheme is illustrated by Fig. 9.2.

*Quotient register*

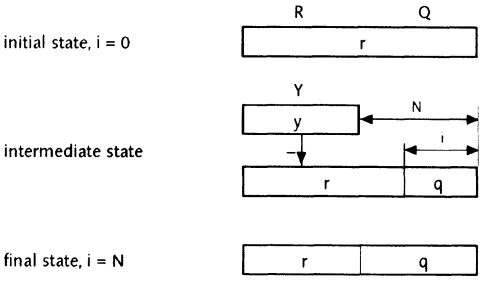



*Fig. 9.2. Division steps using registers R, Q and Y*

The algorithm is shown in terms of the example of dividing X = 122 by Y = 9 (q in italics).

| | | | |
|---|---|---|---|
| Y = | 1001 | | 9 |
| -Y = | 0111 | | -9 |
| | R | Q | |
| r = | 0111 | 1010 | 122 |
| ← | 01111 | 010*0* | step 0 |
| -Y | 10110 | | |
| | 0110 | 010*1* | |
| ← | 01100 | 10*10* | *step 1* |
| -Y | 11011 | | |
| | 1011 | 10*11* | |
| ← | 10111 | 0*110* | *step 2* |
| -Y | 01110 | | |
| | 0111 | 0*110* | |
| ← | 01110 | *1100* | *step 3* |
| -Y | 10101 | | |
| | 0101 | *1101* | *r = 5*, q = 13 |

The shift-subtract step presented above is now programmed in Hercules code:

```
LOD   Q
ADD   Q     q := 2*q
STO   Q
LOD   R
ADDC  R     r := 2*r
```

```
        STO     R
        SUB     Y       r := r-Y
        BCC     L1      branch, if r < Y
        STO     R
        LOD     Q
        ADD     one     q := q+1
        STO     Q
L1      ...
```

This code sequence is repeated 8 times, leaving the quotient in register Q and the remainder in register R. We must point out that when computing with unsigned integers, a subtraction leading to a negative difference must be considered as an overflow. This condition is represented by the C register, resulting in the use of the BCC instruction in the above program.

As in the case of the multiplication algorithm, this program can be shortened as shown below, if the 8 steps are followed by the three instructions labelled by L2. The reader is encouraged to verify this claim.

```
        LOD     Q
        ADDC    Q       q := 2*q
        STO     Q
        LOD     R
        ADDC    R       r := 2*r
        STO     R
        SUB     Y       r := r-Y
        BCC     L1      branch, if r < Y
        STO     R
L1      ...

L2      LOD     Q
        ADDC    Q
        STO     Q
```

## 9.3. Extending the ALU by a Multiplier-Quotient Register

The programs for multiplication and division may be accelerated significantly by adding a dedicated register to the basic architecture, thereby making it possible to represent the add-shift and the shift-subtract steps by a single instruction. The two registers R and MQ are connected by an explicit shift path. The extended ALU structures are shown separately for the multiply and the divide steps. When implementing the scheme, however, the two extensions are merged. Note that now no separate carry register for holding a guard digit is required, as each step is executed by a single instruction in a single interpretation cycle.

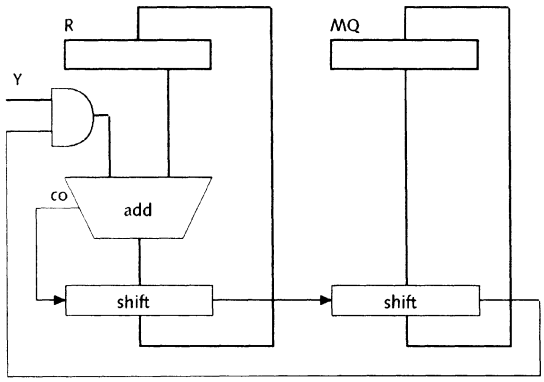



*Fig. 9.3. Circuit implementing the multiply step*

The symbols labelled with "shift" in Fig. 9.3 do not represent any actual circuitry, but merely a displacement of the N parallel signal paths by one position to the right for multiplication or left for division. In practice they are implemented as multiplexers with one input used for loading the registers R and MQ with initial values.

For actual use, the registers must be loadable from some source. This requires additional multiplexers at the register inputs. They are included in the following specification for the multiply-step circuit, which is formulated in terms of N elements and thus is scalable for any word length:

```
MODULE SMult;
   (* s =00: load x, hold y, clear z
      01: hold x, load y, clear z
      10: multiply step*)
   TYPE MulElem;
      IN d, xi, zi, ci, m, a1, a0: BIT;
      OUT x, z, s, co: BIT;
      VAR y, h: BIT;
   BEGIN
      x := REG(MUX(a1: MUX(a0: d, x), xi));
      y := REG(a0, d);
      z := REG(a1*zi);
      h := z - m*y;
      s := h - ci;
      co := (z * m*y) + (h * ci)
   END MulElem;

   CONST N := 8;
   IN s1, s0: BIT;  (*controls*)
      d: [N] BIT;
   OUT z: [2*N] BIT; (*product*)
   VAR M: [N] MulElem;
BEGIN
   M.0(d.0, M.1.x, M.1.s, '0, M.0.x, s1, s0);
   FOR i := 1 .. N-2 DO
      M.i(d.i, M[i+1].x, M[i+1].s, M[i-1].co, M.0.x, s1, s0)
   END ;
   M[N-1](d[N-1], M.0.s, M[N-1].co, M[N-2].co, M.0.x, s1, s0);
   FOR i := 0 .. N-1 DO z.i := M.i.x; z[i+N] := M.i.z END
END SMult.
```

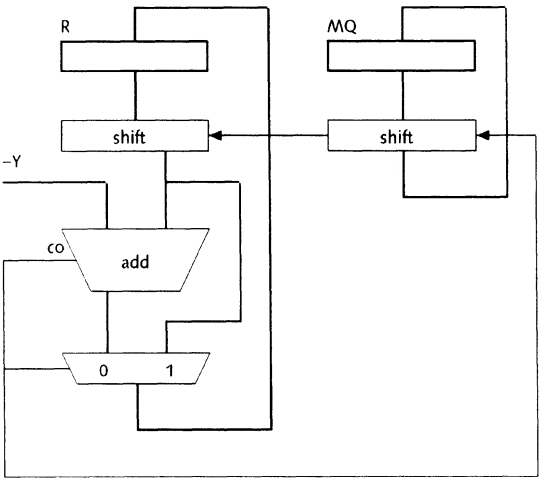



*Fig. 9.4. Circuit implementing the divide step*

As in the case of the multiplier, registers must be made loadable. The divider circuit is specified below, with the divide step shown in terms of an element type.

```
MODULE SDiv;
   (*s = 00: load q, hold y, zero r
      01: hold q, load y, zero r
      10: divide step*)
   TYPE DivElem;
      IN d, ri, qi, ci, s, a1, a0: BIT;
      OUT r, q, co: BIT;
      VAR y, h: BIT;
   BEGIN
      q := REG(MUX(a1: MUX(a0: d, q), qi));
      y := REG(a0, ~d);
      r := REG(a1 * MUX(s: ri, h - ci));
      h := ri - y;
      co := (ri * y) + (h * ci)
   END DivElem;
```

```
    CONST N := 8;
    IN s1, s0: BIT;  (*controls*)
       d: [N] BIT;
    OUT r, q: [N] BIT;  (*remainder, quotient*)
    VAR U: [N] DivElem;
BEGIN
    U.0(d.0, U[N-1].q, U[N-1].co, '1, U[N-1].co, s1, s0);
    FOR i := 1 .. N-1 DO
       U.i(d.i, U[i-1].r, U[i-1].q, U[i-1].co, U[N-1].co, s1, s0)
    END ;
    FOR i := 0 .. N-1 DO q.i := U.i.q; r.i := U.i.r END
END SDiv.
```

**Summary**

Multiplication of two n-bit integers x and y can be achieved either by a purely combinational circuit "in no time", or by adding y repeatedly in x steps. Whereas in the first case, speed is obtained through lavish expenditure of hardware, in the latter simplicity of hardware is bought by affording much time for approaching the result in steps. A compromise between these extremes is common: the use of doubling a value - implementable as a shift - reduces the number of required additions from x to log(x), or at least to n. The operation thereby repeated is called an add-shift step.

A similar compromise is possible for division, reducing the number of required subtractions from x/y to n. The repeated operation is called shift-subtract step.

Both steps can be implemented as single-cycle instructions through slight extensions of the ALU-hardware only. The addition consists of a single register (MQ) which can be considered as extending the accumulator into a register of double length.

# Design of a Computer Based on a Microprocessor

# 10.

**Overview**

The computer of Chapter 8 was constructed in order to explain the fundamentals of computer architecture and to demonstrate a design to its last detail. Computers built in practice use much more highly integrated parts and feature far more sophisticated architectures. They are usually built around one or several microprocessors, each integrating ALU, registers, and control unit. This chapter presents a modest design of this kind. It is based on the processor NS32008 (National Semiconductor, Inc.), and it features the typical configuration with an address and a data bus connecting processor, memories, and peripheral devices.

*Microprocessors*

*Microcontrollers*

Modern semiconductor technology has reached such a high degree of miniaturization that it is possible to place an entire ALU together with a control unit onto a single die. Such highly integrated components are called *microprocessors*. There are even components that also contain a store and interface circuitry for input and output. They typically have a relatively simple ALU-structure and a small store, and they are called *microcontrollers* (see Fig. 10.1).

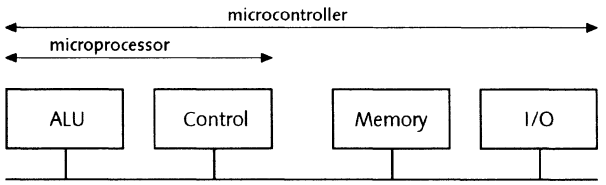



*Fig. 10.1 Microprocessor and microcontroller*

Using a microprocessor, it has become possible to construct an entire computer, featuring all modern programming facilities, with only a few additional components connecting processor and memory by a bus. The basic interface common to all microprocessors is characterized in Fig. 10.2. On its left side it shows as inputs the global signals of

clock and reset, and additionally an interrupt line. The right side represents a typical bus interface, consisting of data, address, and control signals.

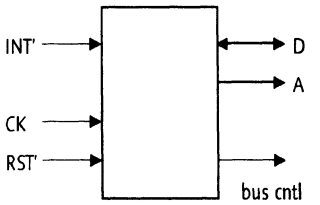



*Fig. 10.2. Typical microprocessor interface*

Subsequently we will demonstrate the design of a simple computer based on a specific microprocessor. We choose the NS32008 (National Semiconductor Corp.) because of its typical structure and its regular instruction set which is convenient for programming. The 32008 is the „smallest" member of an entire family, which all share the same architecture but differ in the widths of their data and address busses, and hence also in their performance. The 32008 features 8 data lines D0 ... D7 and 24 address lines A0 ... A23. From the point of view of the programmer, however, the processor appears as a 32-bit computer, as its internal registers consist of 32 bits each.

*Time-multiplexing*

One of the scarce resources of a chip are its pins. If many data and address lines are present, a solution called time-multiplexing is employed which allows them to be shared. A special signal then indicates whether a line is currently carrying data or an address. In our example, this signal is called *address strobe* (ADS'). Since only 8 data lines are provided, only 8 lines are subjected to time-multiplexing (AD0 ... AD7), whereas the remaining address lines (A8 ... A23) always carry an address value. For the shared lines it is necessary to provide an external register holding the address which appears first on the shared outputs. It is shown in Fig. 10.3, implemented using an 8-bit D-type register (LS574).

*Address strobe*

In principle, address and data lines of the processor may be connected directly to memory components. However, considering that a bus principally serves to make a computer easily extensible, and that the number of components ultimately to be connected is not known a priori, and considering that the driving power of the processor pins

(fan-out) is limited (and usually relatively weak), it is recommendable to provide buffering components. The data buffers are necessarily bi-directional, whereas the address buffers are unidirectional. The direction of the data buffer is determined by a specific processor control output DDIN'.

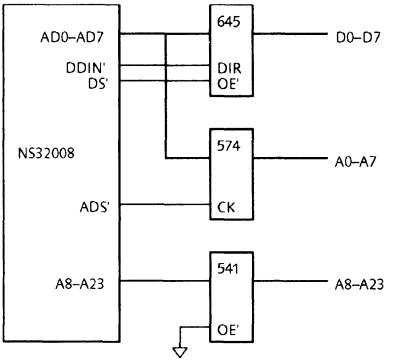



*Fig. 10.3. Microprocessor bus interface*

In order to understand the functioning of this circuit, we need to consider the behaviour of the various signals in time during a memory access cycle (see Figs. 10.4 and 10.5). Every memory cycle consists of 4 phases (subcycles). During the first, labelled T1, ADS' indicates that AD0 - AD7 contain address values, which are latched in the external register. During cycles T2 and T3 memory is accessed, and addresses are stable, as indicated by the data strobe control signal DS'. T4 is a "dead" subcycle letting the circuitry settle for a next cycle.

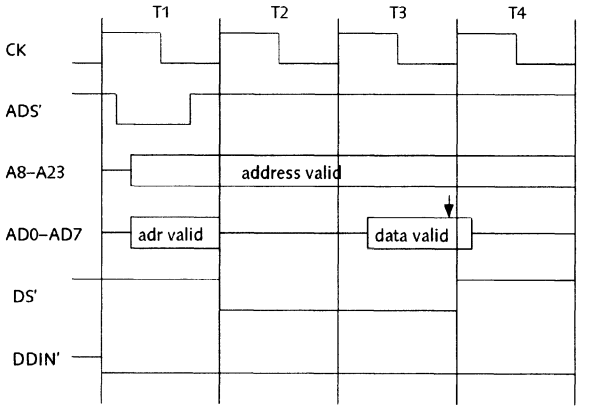



*Fig. 10.4. Read cycle timing*

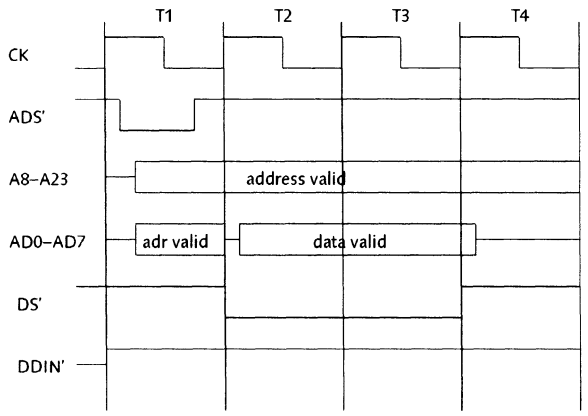



*Fig. 10.5. Write cycle timing*

The clock frequency used with this processor is 10 MHz. It follows that memory components with a cycle time of at most 200 ns must be used, as each subcycle lasts for 100 ns. In order to accommodate slower components, and in particular slower devices, which are addressed on the bus just like memory, a facility is provided to extend an access cycle. It consists of inserting additional T3 subcycles. The processor's state machine senses a specific input signal (RDY') at the end of T2 and T3. If this signal is low at the end of the preceding cycle, the next cycle is again a T3. Thus any device connected to the bus may determine its access time by holding RDY' low as long as needed (see Fig. 10.6).

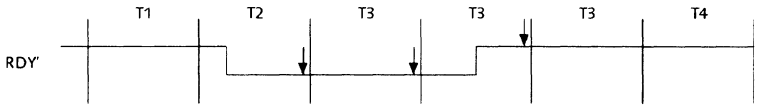



*Fig. 10.6. Extended cycle timing*

In fact the NS32008 features additional possibilities to extend cycles, but we shall not dwell on them in further detail. They are handled by a timing control unit, a component separate from the processor, which also provides control signals for the bus with high fan-out. The essential connections necessary can be read from Fig. 10.7, and further details must be extracted from the pertinent data books.

The extension of the processor cluster, here consisting of the processor, the timing control unit (TCU), and a TL7705 component for gen-

erating a clean reset pulse, to a full computer is straight forward. A configuration is shown in Fig. 10.7 including three SRAM and one ROM components. Since upon reset the processor starts at address 0, the lowest addresses are assigned to the ROM. The address assignment determined by the decoders is:

*Address assignment*

| | | |
|---|---|---|
| 0 - | 00007FFFH | ROM |
| 00008000H - | 0000FFFFH | RAM1 |
| 00010000H - | 00017FFFH | RAM2 |
| 00018000H - | 0001FFFFH | RAM3 |
| FFFFFFF0H - | FFFFFFFFH | UART |

Additionally, an input-output facility is provided in the form of a standard serial communication interface (UART). This device makes use of the extended cycle facility. It also requires a separate clock adapted to the communication standard RS232C, and special voltage levels for the external connection. This is achieved by use of level-shifter components 1488 and 1489 (see also Sect. 12.1).

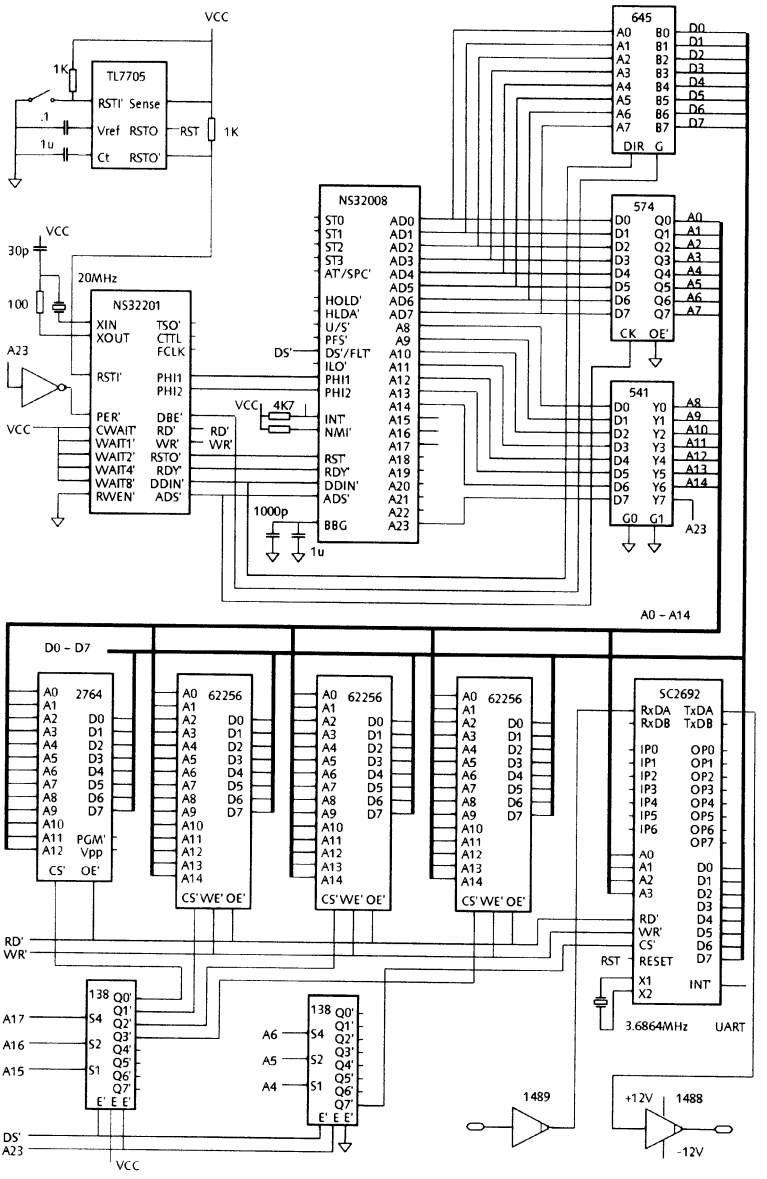



*Fig. 10.7. Computer based on microprocessor NS32008*

Summary of parts used:

| | |
|---|---|
| NS32008 | microprocessor |
| NS32201 | timing control unit (TCU) |
| HM62256 | 32K x 8 SRAM |
| 2764 | 8K x 8 ROM |
| SC2692 | dual UART |
| LS645 | 8-bit transceiver |
| LS574 | 8-bit register |
| LS541 | 8-bit tri-state driver |
| LS138 | 3-to-8 decoder |
| TL7705 | reset signal generator |
| 1488 | RS-232 line driver |
| 1489 | RS-232 line receiver |

# 11. Interfaces Between Asynchronous Units

**Overview**

Up to chapter 9, we have dealt with synchronous circuits only, i.e. with circuits where all registers are controlled by the same, global clock signal. The advantages gained by adhering to synchronicity are considerable, but unfortunately there exist situations where the involvement of several clocks is unavoidable. Here we discuss how independently operating units of a system with autonomous, asynchronous clocks are connected to achieve harmonious cooperation. Evidently, such connections serve to exchange data among the units, and their use must be subjected to rules obeyed by the connected partners. Such rules are called protocols.

## 11.1. The Handshake Protocol

In the preceding chapters, we have concentrated our attention strictly on synchronous circuits. They have some distinctive advantages, and they are much simpler to analyze and verify, because the notion of time has been replaced by the notion of discrete, sequential steps. However, modern computers consist of many modules that operate largely independently and are coupled only during relatively rare instances of data transfer. They are *loosely coupled.* There exist various reasons for letting certain units operate according to their own clock, to let them be *asynchronous* with respect to other units. The trend is to build systems of computers connected via communication lines. Totally disconnected, stand-alone computers are becoming rare. We speak of *distributed systems,* and evidently such networks represent asynchronous systems. We emphasize that the granularity of asynchronous units is quite large, i.e. the components themselves are fairly complex, in themselves synchronous circuits.

*Distributed systems*

This fact allows us to restrict our presentation of problems arising with asynchronous circuits to a specific case, namely the *interface* between two units with the purpose of data transfer.

*Protocols*

Inherently, two asynchronous partners can successfully exchange data only if they agree to a certain convention about engaging in the data transfer. Such conventions are called *protocols.* The most obvious solution of the problem of synchronizing a receiver with a sender is first explained in terms of two respective program fragments. Let the two units be connected by a common variable D (for data). Additionally, two control variables are introduced, one of them being controlled by the sender (req), one by the receiver (ack), and both with initial values 0:

| Sender | Receiver |
|---|---|
| D := x; | REPEAT UNTIL req = 1; |
| req := 1; | y := D; |
| REPEAT UNTIL ack = 1; | ack := 1; |
| req := 0; | REPEAT UNTIL req = 0; |
| REPEAT UNTIL ack = 0 | ack := 0 |

*Request*

*Acknowledgement*

The control signal *req* has the meaning: the data buffer D holds a value to be received (request to receive). And *ack* has the meaning: the request has been satisfied, i.e. the value represented by D has been received (acknowledgement of request). Statements of the form REPEAT UNTIL B represent polling and may imply a delay, thus providing synchronization. Figure 11.1 shows the signal values as functions of time. This process is called a "handshake".

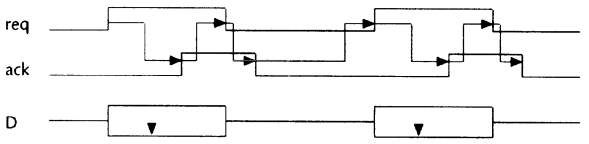



*Fig. 11.1. Handshake with req and ack signals*

*4-phase protocol*

*2-phase protocol*

Each data transfer is accompanied by four control signal changes; the scheme is therefore called a *4-phase protocol,* a term implying an agreement of the two partners to strictly obey the rules when communicating. A slightly simpler and faster variant is the *2-phase protocol* based on the invariant: *req* ≠ *ack* implies that D holds a value to be transferred (see Fig. 11.2). From the point of view of hardware, both protocols require the same resources.

| Sender | Receiver |
|---|---|
| D := x; | REPEAT UNTIL req # ack; |
| req := ~req; | y := D; |
| REPEAT UNTIL ack = req | ack := req |

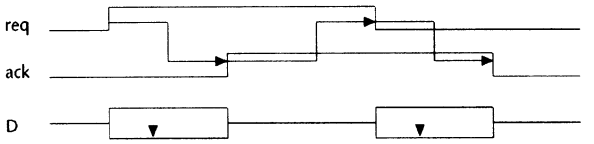



*Fig. 11.2. 2-phase protocol*

*SCSI*

The 4-phase protocol is used by the widespread SCSI interface standard (Small Computer Systems Interface). It requires two control lines in addition to the data line; hence the control lines represent a heavy "overhead". This overhead is reduced, if multiple, parallel data lines are provided. SCSI specifies 9 data lines (8 plus 1 for parity), and thus the "overhead" is only 2 out of 11 lines. SCSI is an *asynchronous, parallel interface.*

The scheme considered here represents a uni-directional data transfer from a sender to a receiver, i.e. a point-to-point connection (Fig. 11.3). In the presented constellation, the transmitter issues requests to the receiver; they are *read requests.* If the receiver is to be able to issue requests too, they are requests to the transmitter, i.e. *write requests.* In this case the meaning of the *req* signal is: request to send data is pending; and that of *ack* is: data D are valid and to be received. The only difference in the timing diagram with respect to Fig. 11.1 is that D is valid while *ack* rather than *req* is active.

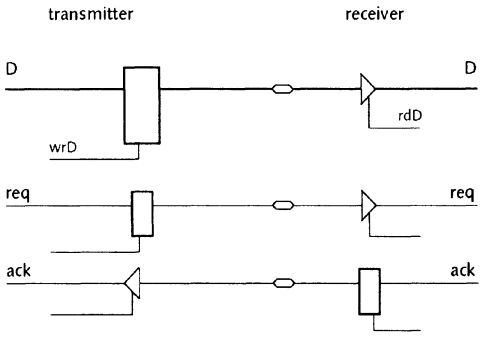



*Fig. 11.3. Point-to-point connection with req/ack protocol*

But frequently, a bi-directional connection is desired. In this case, the number of lines (wires) is reduced by sharing them for both directions, implying that at any time only one direction is operational. If such a scheme is desired, one might as well generalize to the case with not only two, but an arbitrary number of data sources, thereby obtaining a bus. Indeed, SCSI allows up to 8 partners to be connected and represents a bus system (see Fig. 11.4). It uses open collector drivers. Receiver/transmitter pairs are packed into a single unit (chip). In addition to the 9 data lines and the *req* and *ack* lines, it features 7 additional control lines, mainly used for the higher-level protocol subdividing a transaction into phases, but also for arbitrating in the case of simultaneous bus requests.

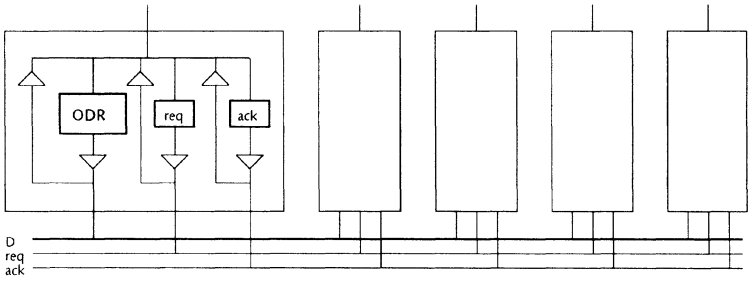



*Fig. 11.4. Simplified SCSI bus*

## 11.2. Processor-Bus Interfaces

Often, interfaces between processor busses and asynchronous devices use a somewhat simpler scheme. It is based on a single variable *rdy* instead of two variables *req* and *ack*. Its protocol is described by the

following program segments, where *rdy* has the meaning: variable D holds a value to be accepted.

| Sender | Receiver |
|---|---|
| D, rdy := x, 1; | REPEAT UNTIL rdy = 1; |
| REPEAT UNTIL rdy = 0 | y, rdy := D, 0 |

A new problem arises through the fact that the common variable *rdy* can be changed by both the transmitter and the receiver. With relief, however, we notice that the transmitter assigns 1 (sets) and the receiver assigns 0 (resets) only. Furthermore, they obey the preconditions *rdy* = 0 for assigning 1 and *rdy* = 1 for resetting. Hence, the SR-latch (see Chapter 3) is exactly the circuit to be employed for representing the variable *rdy*. The basic circuit for a processor bus interface representing an output port is shown in Fig. 11.5. Its detailed timing diagram is given by Fig. 11.6. *rdy* = 1 implies that the data buffer is ready to be read, *rdy'* = 1 implies that the data buffer is ready to be loaded. The data buffer is typically implemented by D-latches.

*SR-latch*

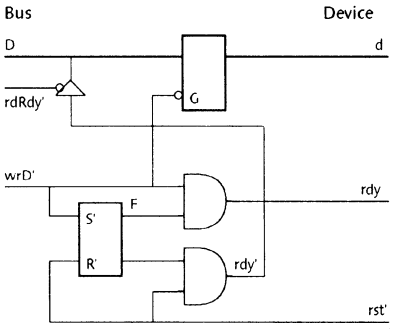



*Fig. 11.5. Processor bus output port*

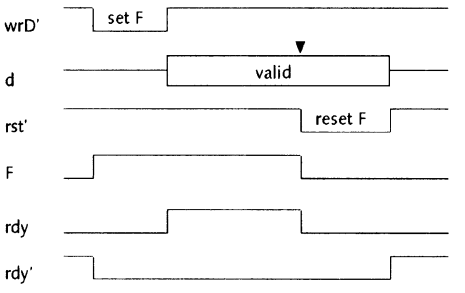



*Fig. 11.6. Timing diagram of port signals*

A bus interface representing an input port is almost identical, except that the data hold register is considered as part of the device circuitry, and therefore clocked by the respective global clock (see Fig. 11.7).

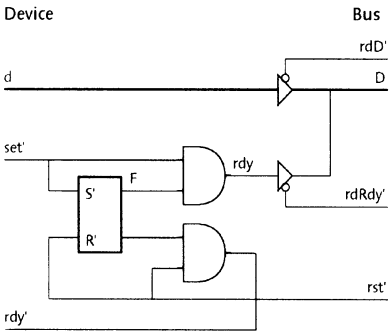



*Fig. 11.7. Processor bus input port*

The control pulses *wrD'* and *rdRdy'* in Fig. 11.5, and *rdD'*, *rdRdy'* and *rst'* in Fig. 11.7 are derived from the control signals WR' and RD' of the bus and from address lines through an appropriate decoder. The additional *and* gates at the output of the SR-latch are necessary, because processor busses typically specify that the data lines hold a valid value upon the rising edge of the *WR'* control signal, but not necessarily before. Therefore, the *rdy* signal must not become active before the end of the write strobe *wrD'*. Note that *rdy'* is *not* the same as *~rdy*.

## 11.3. Adding an I/O Interface to the Hercules Computer

We are now ready for the completion of the Hercules computer (Chapter 8) by adding an input and an output port. The latter follows the scheme presented by Fig. 11.5, the input port that of Fig. 11.7 with the addition of an explicit output register *do*. The extension is reflected by the addition of two instructions for reading the data from the input latch (READ), and for storing data in the output register (WRITE). Furthermore, two cases are added to the set of conditions available for branch instructions. The new instructions are BIC (branch if input latch clear) and BOC (branch if output register clear). READ and WRITE belong to the class of data instructions; we choose their opcode values to be 28H and 0E8H respectively. They activate the control signals

```
selin := ~IR.7 * ~IR.6 * IR.5 * ~IR.4 * IR.3      READ
out := IR.7 * IR.6 * IR.5 * ~IR.4 * IR.3 * PH.3   WRITE
```

The input port appears as an additional data source of the accumulator, like the *and, or, xor,* and *add* functions. *out* is used as enable signal for the output register *do* and is active during phase 3 only. The specification of the ALU slice is extended by the declaration

```
OUT do: BIT
```

and by the statements

```
z := seland * (x*y) + ... + selin * LATCH(w0', D);
do := REG(out, R)
```

w0' is the external signal which opens the input latch to accept data D. For representing the states of the input latch and the output register, two SR-latches called *inflg* and *outflg* are introduced. Their specifications follow Sect. 11.2.

```
inflg := SR(w0', ~selin);
ine' := inflg * w0';          input latch not empty
ine := ~inflg * ~selin;       input latch empty
outflg := SR(~out, w0');
oute' := outflg * ~out;       output register not empty
oute := ~outflg * w1'         output register empty
```

*Inflg, outflg*

w1' is the external signal which resets *outflg* after the data in the output register had been fetched. The condition for branching now obtains two additional terms involving the two flags:

```
cond := ~(IR.7*oute + IR.6*ine' + IR.5*C + IR.4*alu.7.zo + IR.2*alu.7.R)
```

And this completes the addition of a simple I/O facility to the Hercules computer, shown with its external connections in Fig. 11.8.

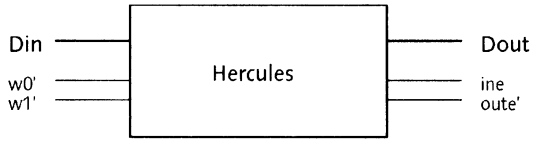



*Fig. 11.8. Hercules computer of Chapter 8 with I/O interface*

**Summary**

Interfaces between asynchronously operating parts of a system consist of a data buffer and state variables indicating whether or not the buffer holds valid data. The most common manner of synchronizing partners for data transmission is the so-called *handshake* protocol. It involves, in addition to the data signals, a *request* and an *acknowledge* signal. This "overhead" of control signals is justified only if several data lines exist, i.e. in the case of parallel data transmission, of a parallel interface. A well-known example is the SCSI-Standard.

Interfaces between processors and peripheral devices connected by a bus usually employ a similar scheme. But instead of featuring two separate flags for request and acknowledge signals, they use a single flag, which is set by one and reset by the other of the partners.

Asynchronous operation should be avoided whenever possible, as the analysis of asynchronous circuits is much more intricate than that of synchronous systems. The step from synchronous to asynchronous circuits is comparable to that from uniprogramming to multiprogramming in the realm of software. In particular, experimental testing becomes inherently inadequate, because process constellations are irreproducible.

# Serial Data Transmission

# 12.

**Overview**

Whereas data are typically transmitted over parallel wires within computers, they are transmitted serially over a single wire between computers. Shift registers are used to serialize and deserialize data streams. If serializing and deserializing shifters are controlled by the same clock signal, we speak of synchronous, otherwise of asynchronous transmission. In the former case, the clock is either transmitted on a separate wire, or it is encoded and mixed with the data signal.

## 12.1. Introduction

The subjects and technologies of data processing (computing) and data transmission (communication) have become inseparably intertwined. There hardly exist "stand-alone" computers any more, and communications equipment invariably contains components that resemble computer circuitry. It is therefore appropriate to end with a chapter on data transmission techniques.

The hardware involved in the transmission of digital signals consists of three entities, namely an interface between the sending computer and the line called the *transmitter*, the *transmission line* itself, and an interface between the line and the receiving equipment called the *receiver*. Here we shall concentrate our attention on the transmitter and receiver and assume the transmission line to be a straightforward connection. In reality, however, signals travelling over long lines are weakened and distorted. These phenomena are explained by transmission line theory. They depend on the frequencies of the transmitted signals. Here we merely assume that distortion and loss can safely be ignored for signals below a certain limiting frequency $f_{lim}$, measured in Hz (hertz), which is also called the bandwidth of the line.

*Bandwidth*

Given a line with a bandwidth $f_{lim}$, what is the maximum data rate, the maximum number of bits per second (b/s) that can be transmitted over this line? In order to answer this basic question, consider the signal where consecutive bits alternate between 0 and 1. Let this square

*data rate*

wave *d* be approximated by a sinusoidal signal *s*. At the receiving end, the digital signal can be recaptured by sampling s at the midpoints of each so-called *bit-cell*, as indicated in Fig. 12.1. Evidently, s has a period twice as long as the bit-cell. Hence, we conclude that the line's bandwidth must be (at least) half the clock frequency of d.

*Bit-cell*

$f_{lim} = f_{ck} / 2$

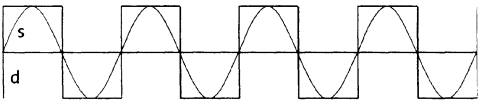



*Fig. 12.1. Bandwidth = Half data rate*

Within computers, data are usually transmitted in parallel, with a data path width of 8, 16, 32, or 64 lines (bits). This is uneconomical, if transmission is to be over larger distances, because long lines are costly, or it is even impossible, if existing facilities, such as telephone lines or radio channels are the chosen transmission medium. The solution lies in serializing the data in the transmitter and deserializing them in the receiver (Fig. 12.2).

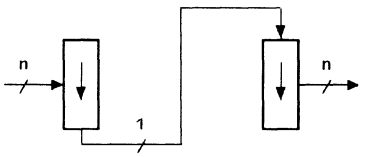



*Fig. 12.2. Shift registers for serializing and deserializing data*

## 12.2. Synchronous Transmission

The principal question arising with the configuration of Fig. 12.2 is: When should the line be sampled by the receiver in order to recapture the original digital signal? The simplest solution is to clock the receiving shift register with the same clock as the transmitting shift register, i.e. let the two registers be synchronous. This implies that the transmitter's clock must be available to the receiver, giving rise to a second wire for the clock, parallel to the data wire. This solution is typically chosen where wires are moderately long, such as between computers and nearby peripheral devices, and in so-called embedded equipment. Several industrial standards have emerged which are based on this

arrangement, e.g. National Semiconductor's *Microwire*, and Motorola's *SPI* (Serial Peripheral Interface). They permit simultaneous data transfer in both directions by postulating a third wire as shown in Fig. 12.3. The circuit generating the clock signal is called the *master*, the other the *slave*.

*Master/slave.*

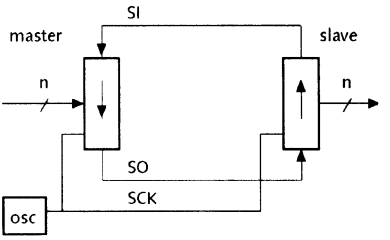



*Fig. 12.3. Bidirectional transmission with separate clock line*

This scheme works well if signal delays over the lines are negligible compared with a clock period. If the master is used as sender only, i.e. if SI is not involved, then this condition can be relaxed, and it is important only that SO and SCK have equal propagation delays in order to avoid clock skew.

We now proceed to develop a master transceiver in detail. Its interface to the driving computer consists of the 8-bit data bus, a single address line, a write-strobe (trig') and a read-strobe (rd'), providing the following interactions (*base* is the I/O address of the transceiver).

```
Sending a byte:    PUT(base, x);                  trigger, send x
                   REPEAT UNTIL BIT(base+4, 0)    wait until done

Receiving a byte:  REPEAT UNTIL BIT(base+4, 0)    wait until done
                   GET(base, x)                   fetch received byte x
```

Its body consists of the shift register (S), its heart of a state machine controlling the transmission process. An almost obvious solution is to break up the state machine into two parts: a single state bit (*run*) indicating whether transmission is in progress, and a counter (q) indicating the number of the bit being transmitted. Assuming byte-wise operation, a 3-bit binary counter suffices. *run* becomes active upon a trigger from the interfaced computer. At this time, the shift register S is loaded from the computer's bus. *run* returns to the inactive state after the last bit had been shifted out. Since the triggering computer

and the transmitter have their own clocks, the asynchronous interface as presented in Sect. 11.2 is our likely choice, the SR-latch representing the signal *run*. However, this is unacceptable, because unfortunate clock coincidences might cause the state machine to advance before the data are properly latched in the shifter. The remedy lies in the introduction of a so-called *synchronizer*, a flipflop which delays the asynchronous trigger until the next cycle of the transmitter's own clock. Often the receiver is clocked by the inverted transmitter clock, thus obtaining a delay of 1/2 bit time, and thereby increasing the line's reliability and robustness against noise.

*Synchronizer*

With these explanations in mind, consider the signals in Fig. 12.4, shown progressing through a full transmission cycle:

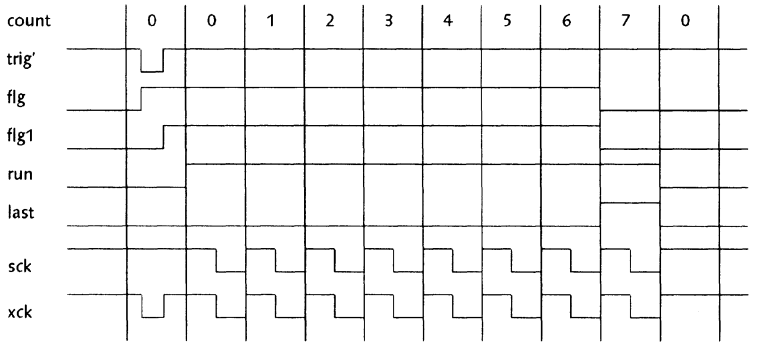



*Fig. 12.4. Timing diagram of transmitter signals*

We can now specify the transmitter circuit by the following Lola program:

```
MODULE Transceiver;
   IN trig', rd', A2, ck, SI: BIT;
   INOUT D: [8] TS;
   OUT SO, SCK: BIT;
   VAR xck, sck: BIT;
         last, flg, flg1, run: BIT;
         q, c: [3] BIT; (*counter*)
         S: [8] BIT;  (*shift register*)
BEGIN
   q.0 := REG(ck: q.0 - run); c.0 := q.0 * run;
```

```
    q.1 := REG(ck: q.1 - c.0); c.1 := q.1 * c.0;
    q.2 := REG(ck: q.2 - c.1); c.2 := q.2 * c.1;
    last := q.0 * q.1 * q.2;
    flg := SR(trig', ~last); flg1 := flg * trig';
    run := REG(ck: flg1);  (*synchronized flag*)
    sck := ~run + ck; xck := trig' * sck;

    S.0 := REG(xck: MUX(run: D.0, SI));
    D.0 := ~rd' | MUX(A2: S.0, ~flg1*~run);
    FOR i := 1 .. 7 DO
        S.i := REG(xck: MUX(run: D.i, S[i-1]));
        D.i := ~rd' | S.i
    END ;
    SO := S.7; SCK := ~sck
END Transceiver
```

The trigger signal also loads the shift register. Note that here we use - against our own recommendation in Chapter 3 - a gated clock *xck*, combining the trigger with the shift clock *sck*. This is done to keep the circuit simple, and is unproblematic in this case. Note also that by connecting the shift register to the data bus D via tri-state gates, the circuit can be used as transmitter as well as receiver. It is therefore called a *transceiver.*

*Transceiver*

As already mentioned, this setup is used by several industrial standards, and therefore devices are available for several purposes complying with the standard. For example, many microcontrollers incorporate such a transceiver, which allows a cheap connection to, for example, analog-to-digital converters (ADC), or digital-to-analog converters (DAC). Such a configuration is shown in Fig. 12.5.

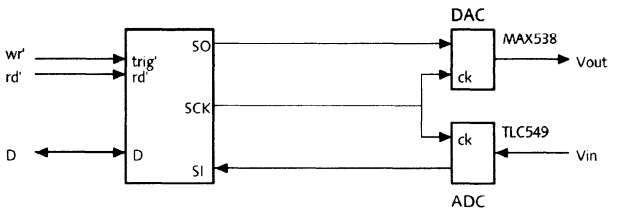



*Fig. 12.5. Transceiver connected with DAC and ADC*

*Clock encoding*

*Modulation*

The inherent drawback of this scheme is the requirement for a separate clock wire. This is not acceptable if distances are long. Then the solution lies in combining, mixing the data and clock signals into a single signal by suitable encoding. There exist several proven *encoding techniques.* Here we shall describe only one of them, namely *frequency modulation* (FM). It necessitates, as do all encoding methods, a *modulator* at the transmitter's output and a *demodulator* at the receiver's input. The latter is also called a *data separator.*

The basic idea of frequency modulation is to use different signal frequencies for the transmission of zeroes and ones. Typically, one of them is twice the other. The encoded signal can be generated by the following recipe: The line signal s changes (from 0 to 1 or vice versa) at the beginning of each bit cell. It also changes in the middle of cells corresponding to a zero (d = 0). An example is shown in Fig. 12.6.

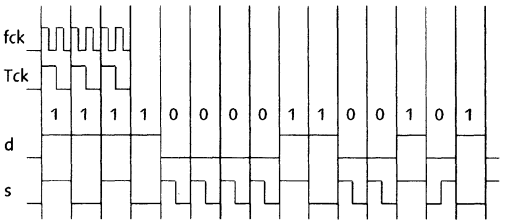



*Fig. 12.6. FM-signal s encoding digital signal d*

The price for this method is, as is to be expected, a higher line bandwidth. In order to transmit n bits per second, a bandwidth of n Hz is required, as opposed to n/2 for a straight digital signal. A modulator circuit is shown in Fig. 12.7. It uses a doubled clock rate for the two registers, thus clocking the output register both at the beginning and in the middle of every bit cell. s changes (toggles) whenever t = 1.

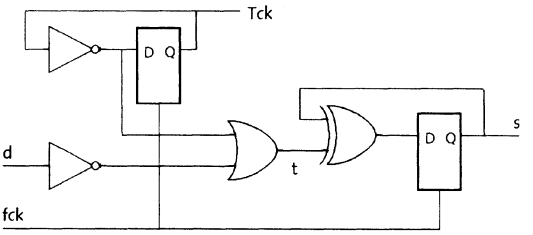



*Fig. 12.7. Digital frequency modulator*

The separation of data and clock signals at the receiver's end is slightly more complicated. The received signal, which is likely to be distorted (not an ideal square wave) must first be restored. This is done by sampling the input at a rate considerably higher than the transmission rate. This is called *oversampling*, and the sampling rate is typically 16 times the bit rate. We recall that this clock can never be absolutely exactly equal to the (multiple of the) transmitter frequency. The task of the separator is therefore also to recover the transmitter clock *Rck*.

*Oversampling*

The essence of the data/clock separator is explained as follows: The oversampling clock *sck* runs a counter. A transition of the input (t = 1) detected while the counter value is, say between 4 and 11 (a = 1), i.e. near the cell's middle, indicates a 0 data bit (c). A transition at counter value not less than 12 (~a) indicates a bit cell's end (b), and serves to generate the restored clock. Normally, such a transition occurs at counter value 15. Any other value is due to clock deviation. Detection of such a „clock transition" causes the counter to be reset to 0. A demodulator using a counter C with synchronous clear and a decoded signal *a* for indicating counter value < 12 is shown in Fig. 12.8.

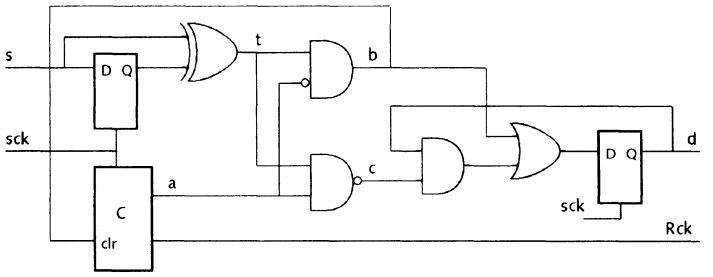



*Fig. 12.8. Data/clock separator*

A cell-end transition of s, indicated by t being high, marks the beginning of the next bit cell and sets output d high. If a mid-cell transition is detected, i.e. t high while a high, c becomes low and resets d to low. Otherwise d remains high until the cell-end is reached, when the restored clock *Rck* (q.2 of counter C) causes d to be latched in the receiver. We point out that this circuit operates synchronously at the oversampling rate, which is typically quite high.

By using a data separator, the transmitter's clock is duly recovered at the receiver's end, and we therefore speak of synchronous transmis-

sion, just as if the clock had travelled on a separate wire. Arbitrarily long sequences of bits can be transmitted reliably. Frequency modulation is used by several standards, for example by *SDLC* (synchronous data link control), IBM's *Bisync,* and *Ethernet.* The latter operates at 10 Mb/s (requiring a line bandwidth of 10 MHz). At frequencies above about 1 MHz digital modulators and demodulators are replaced by circuits operating with analog signals, because the oversampling frequencies would become quite high.

*SDLC*
*Ethernet*

Recently, much effort has been spent in devising sophisticated encoding techniques which allow one to increase the transmission rate for a given bandwidth, approaching the theoretical limit of 2 bits/hz. There is considerable economic incentive behind this endeavour, because the connections of individual customers, essentially telephones in homes, to central exchange offices represent one of the largest single investments, comparable to interstate highway systems. As these lines are fixed, there is a substantial benefit to be gained from increasing the data rate over these lines with given, limited bandwidth. What made the substantial improvements possible was the advent of powerful digital signal processors (DSP).

## 12.3. Asynchronous Transmission

The complications of modulator and demodulator could be avoided if absolutely precise clock oscillators were available. This is, of course, not so. However, crystal oscillators are available with deviations guaranteed to be less than $10^{-5}$. Therefore, bit sequences of limited length (so-called *packets*) can be transmitted reliably without clock encoding, i.e. with unsynchronized clocks. This technique is called *asynchronous transmission.* Its advantage is, beside simpler circuits, a higher transmission rate with lines of the same bandwidth. In fact, a factor of 2 can be gained over simple frequency modulation. The method as such is quite old, and is used by the widely established RS232 standard, which specifies a small, fixed packet length, such that each packet represents a single byte.

*RS232*

Nevertheless, there remains one problem: Not only must oscillator frequencies be the same, but also the phase must be known to the receiver. It is necessary to determine the exact times for sampling the

incoming signal. The simple solution lies in letting each packet be preceded by a single *start bit* triggering the receiver. If the idle value of a line is defined to be 1, then the start bit value is to be 0. Consequently, each packet is also to be followed by a *stop bit* with idle value 1 in order to make sure that the start bit of the next packet can be recognized as a signal transition. The RS232 standard specifies variants with several stop bits; they merely assume the role of giving the receiver enough time to process or store the received byte. The format of a packet is shown in Fig. 12.9.

*Start bit*

*Stop bit*

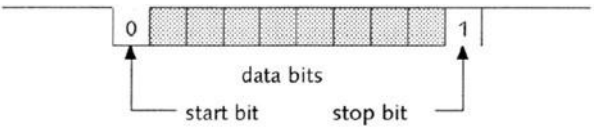



*Fig. 12.9. RS-232 packet format*

A component which integrates a transmitter and a receiver is called a *universal, asynchronous receiver and transmitter* (UART, see also Chapter 10). The attribute „universal" stems from the possibility to select by program control several parameters of the standard. They include the transmission rate, the number of data bits per packet (usually 8), the number of stop bits (usually 1), and whether a parity bit is to be appended to the data bits and checked. Among the specified transmission rate are

*UART*

110, 300, 600, 1200, 2400, 4800, 9600, 19200 b/s

*Standard transmission rates*

Furthermore, the standard specifies the voltage ranges admissible for representing the bit values. They are -5 ... -15V for zeroes, and +5 ... +15V for ones. These specifications stem from the era preceding TTL technology and unfortunately require supply voltages different from the standard TTL supply of 5V. They have given rise to special parts (chips) capable of generating the required positive and negative supply voltages from the single 5V supply always available. Parts which convert TTL signals to the voltage ranges prescribed by RS-232 are called *line drivers* and *line receivers* and, due to their voltage level adaptation, also *level shifters*. A circuit consisting of a UART and level shifters is shown in Fig. 12.10.

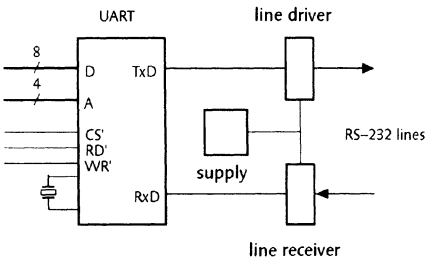



*Fig. 12.10. RS-232 interface with UART*

The address lines of the UART (typically 3 or 4) serve to select various internal registers, among them the data buffers, but also a status register, and state registers holding the various, selected transmission parameters.

*RS-485*

A more modern standard for transmission lines is RS-422 (and, almost identical, RS-485). It specifies for each line two wires, carrying symmetric signals in order to reduce sensitivity against noise. It also reduces the influence of differences in ground potential at the line's ends. The line can be implemented by a twisted wire pair, allowing bit rates of up to several MHz.

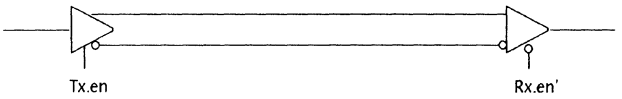



*Fig. 12.11. RS-422 (RS-485) line with driver and receiver*

Again, the interface to the driving computer consists of the 8-bit data bus D, address lines, a write-strobe WR' and a read-strobe RD'. Program statements to send and receive a byte are similar to those in the preceding section; *base* is the I/O address of the UART.

```
Sending a byte:    REPEAT UNTIL BIT(base+4, 1)    wait until Tx.rdy
                   PUT(base, x);                  send x
Receiving a byte:  REPEAT UNTIL BIT(base+4, 0)    wait until Rx.rdy
                   GET(base, x)                   fetch received byte x
```

We now turn our attention to the design of a transmitter and confine the design to a single clock rate, a fixed packet length (8), and a single stop bit. Under these assumptions, the circuit is quite similar to the transmitter in Sect. 12.2, with the exception of a somewhat more complex state machine.

In view of the necessity of oversampling in the receiver, and thereby of a higher clock rate, and the wish to use the same basic clock for both transmitter and receiver, a frequency divider is needed to generate the bit clock. It is implemented by a 4-bit binary counter C with enable. Choosing the standard RS-232 rate of 9600 b/s and an oversampling factor of 16, the clock frequency becomes 16*9600 = 153600 Hz. The signal is typically obtained from a crystal oscillator with the standard frequency 3.6864 MHz, and by dividing its output frequency by 24. The divider is called a *prescaler.*

*Clock counter*
*Bit counter*

Counter C, the *clock counter,* is followed by the 4-bit *bit counter* B, whose value represents the number of the bit being transmitted. B is in fact the state machine adapted from Sect. 12.2. By „followed" is meant that the carry output of C serves as carry input (enable) of counter B. The entire circuit is thus synchronous and driven by the prescaler's output.

The system is set in motion by a trigger, typically an external WR-strobe, resetting an SR-latch which constitutes the interface to the driving computer (see Sect. 11.3). The latch is set by counter B indicating that the last bit is being transmitted. These explanations lead to the block schematic of Fig. 12.12, which is based on the signals shown in Fig. 12.13.

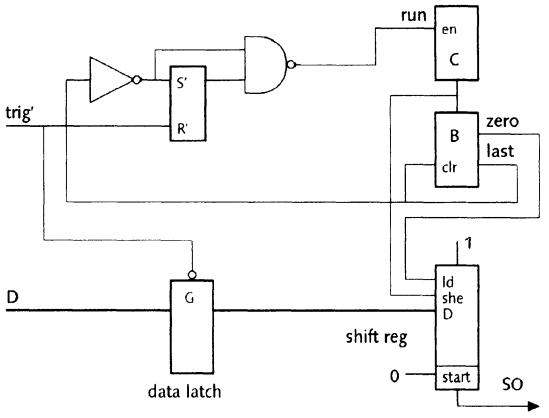



*Fig. 12.12. Block schematic of transmitter*

The shift register is extended by one element representing the start bit. When the shifter is loaded with data, the start bit register is loaded with 0.

```
zero := ~B.q.0 * ~B.q.1 * ~B.q.2 * ~ B.q.3;
last := B.q.1 * B.q.3;  (*10*)
flag := SR(~last, trig');
run := ~(flag * ~last)
```

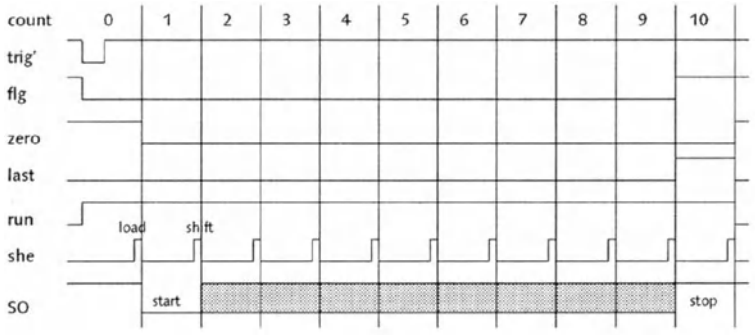



*Fig. 12.13. Transmitter signals*

The detailed specification of the transmitter is given at the end of this section. A few additional remarks referring to this specification are appropriate at this point:

1. A *rdy* (= ~run) signal is provided at the interface. It must be 1 whenever a trigger is issued.
2. The data are latched at the bus input. This is necessary, because the shift register accepts data at the end of the first bit time (C = 15), by which time the data may have disappeared at the bus input. We assume that the trigger pulse which enables the data latch is shorter than a bit time.
3. Data are inverted before they reach the shifter. This is in consideration of the fact that many register parts provide a reset signal, but no set signal for startup. Since in this case the start bit register would be 0 upon system reset, the line would indicate the start of a transmission, activating the receiver. In order to prevent this from happening, an inverter is placed between the shifter and the output SO, requiring in turn the inverters at the shifter's input.

4. The use of a decoded signal (last) as input to the SR-latch represents a potential hazard. If there are only two terms, as is the case here, glitches are often small enough to be negligible.

The corresponding receiver is similarly organized, in the sense that it also consists of a shift register Q, a clock counter C, and a bit counter B, which, however, may disregard stop bits. It also contains the obligatory interfacing SR-latch. Here, the latch is set during the last bit time, and reset when the shift register is read out. Hence, the latch indicates whether a next byte had been received.

Whereas the transmitter is triggered by a pulse from the interface, the receiver is triggered by the start bit from the serial input SI. The receiver's block schematic and the receiver's signals are shown in Figs. 12.14 and 12.15 respectively.

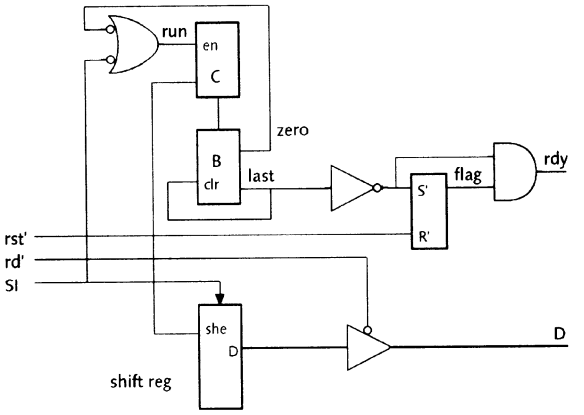



*Fig. 12.14. Block schematic of receiver*

```
zero := ~B.q.0 * ~B.q.1 * ~B.q.2 * ~ B.q.3;
last := B.q.3;  (*8*)
run := ~SI + ~zero;
flag := SR(~last, rst');
rdy := flag * ~last
```

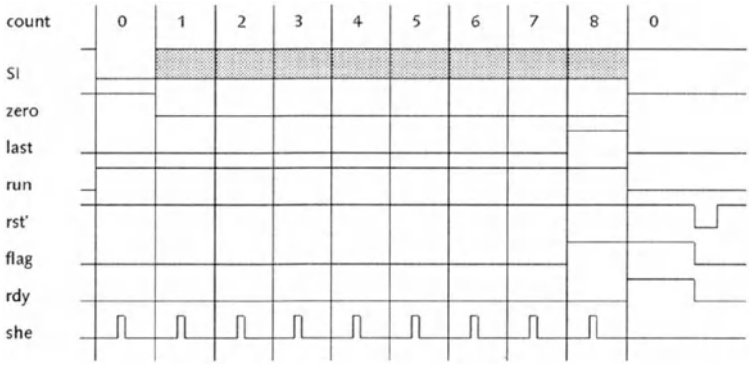



*Fig. 12.15. Receiver signals*

An important detail concerns the shift enable signal, which determines when the serial input is sampled. It is chosen such that sampling occurs in the middle of the bit period, ensuring a better transmission reliability. This is achieved by using the C-counter value 7 as shift enable (rather than 15), i.e. *she* = ~C.q.3 * C.c.2.

A further, rather subtle detail has to be considered for generating the signal *zero*. It is defined as expected, but with the additional term ~C.q.3, i.e. as

zero := ~C.q.3 * ~B.q.0 * ~B.q.1 * ~B.q.2 * ~B.q.3

To understand the reason for this seemingly superfluous addition, consider the timing during the reception of the start bit as shown in Fig. 12.16. Remember that receiver and transmitter clocks are independent, asynchronous, and assume that by chance both coincide, or rather that the transmitter clock precedes by a small amount, in fact by an amount which is less than a register's setup time, and perhaps more when considering line signal distortions.

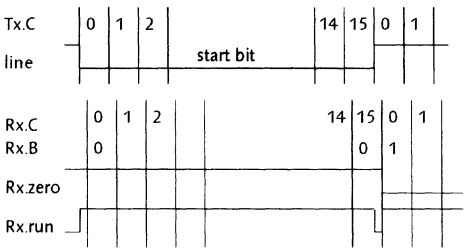



*Fig. 12.16. Receiver missing start bit*

Because the receiver's clock counter (barely) missed the zero transition of the line signal SI, it starts counting late. As a consequence, SI goes high (assuming D.0 = 1) before the clock advances counter C from 15 to 0. Since at that time the bit counter B has not reached the value 1 yet, *zero* is still high, and *Rx.run* (= ~SI + ~zero) goes low (see spike in Fig. 12.16) which stops the clock counter and thereby also the bit counter. The entire machinery comes to a premature halt. The additional term ~C.q.3 lets *zero* go low early enough to prevent this „coincidence" from happening. This is a typical situation encountered in asynchronous systems that is easily overlooked and leads to spurious malfunctioning.

We recognize the analogy of synchronous versus asynchronous circuit design to sequential versus concurrent programming. Testing asynchronous circuits is as questionable as that of concurrent programs. Correctness can only be demonstrated convincingly by detailed analysis. While testing may be necessary, it cannot be sufficient.

```
MODULE UART;
  TYPE Prescaler;
    IN fck: BIT;
    OUT ck: BIT;
      (* Tx and Rx clock: 3686.4 / 24 = 153.6 KHz = 16 * 9600 Hz*)
    VAR H: [5] BIT;
  BEGIN
    H.0 := REG(fck: H.0 - H.1);
    H.1 := REG(fck: ~H.0);
    H.2 := REG(fck: H.2 - H.0);
    H.3 := REG(fck: H.3 - H.0*H.2);
    H.4 := REG(fck: H.4 - H.0*H.2*H.3);
    ck := H.4
  END Prescaler;

  TYPE Counter;
    IN e: BIT;
    OUT q, c: [4] BIT;
  BEGIN q.0 := REG(q.0 - e); c.0 := q.0 * e;
    FOR i := 1 .. 3 DO q.i := REG(q.i - c[i-1]); c.i := q.i * c[i-1] END
  END Counter;
```

```
TYPE CounterClr;
   IN e, clr: BIT;
   OUT q, c: [4] BIT;
BEGIN q.0 := REG(q.0 + clr - e); c.0 := (q.0 + clr) * e;
   FOR i := 1 .. 3 DO
      q.i := REG((q.i + clr) - c[i-1]); c.i := (q.i + clr) * c[i-1]
      END
END CounterClr;

TYPE Receiver;
   IN rst', in: BIT;
   OUT rdy: BIT;
      Q: [8] BIT;
   VAR zero, last, flag, run, she: BIT;
      C: Counter;  (*clock counter: 153.6 / 16 = 9.6 KHz*)
      B: CounterClr;  (*bit counter, 0 .. 9*)
BEGIN C(run); B(C.c.3, last);
   zero := ~C.q.3 * ~B.q.0 * ~B.q.1 * ~B.q.2 * ~B.q.3;
   last := B.q.3;  (*8*)
   flag := SR(~last, rst');
   run := ~in + ~zero;
   rdy := flag * ~last;
   she := ~C.q.3 * C.c.2;  (*shift enable in middle of bit period*)
   Q.7 := REG(she, in);
   FOR i := 0 .. 6 DO Q.i := REG(she, Q[i+1]) END
END Receiver;

TYPE Transmitter;
   IN trig: BIT;
      d: [] BIT;
   OUT out, rdy: BIT;
   VAR zero, last, flag, run, she: BIT;
      C: Counter;  (*clock counter 153.6 / 16 = 9.6 KHz*)
      B: CounterClr;  (*bit counter, 0 .. 9*)
      Q: [8] BIT;  (*shift register*)
```

```
BEGIN C(run); B(C.c.3, last);
        zero := ~B.q.0 * ~B.q.1 * ~B.q.2 * ~B.q.3;
        last := B.q.1 * B.q.3;  (*9*)
        flag := SR(~last, ~trig);
        run := ~(flag * ~last);
        rdy := ~run;
        she := C.c.3;
        out := ~REG(she, MUX(zero: Q.0, '1));
        FOR i := 0 .. 6 DO
            Q.i := REG(she, MUX(zero: Q[i+1], LATCH(trig, ~d.i)))
        END ;
        Q.7 := REG(she, MUX(zero: '0, LATCH(trig, ~d.7)))
    END Transmitter;

    IN RD', WR', A2: BIT;  (*controls*)
        CK: BIT;  (*3.6864 MHz clock*)
        in: BIT;  (*serial input*)
    INOUT D: [8] TS;
    OUT out: BIT;  (*serial output*)
    VAR ins: BIT;
        P: Prescaler;
        Rx: Receiver;
        Tx: Transmitter;
    CLOCK P.ck;
BEGIN P(CK);
    ins := REG(in);  (*synchronizer*)
    Rx(WR'+ ~A2, ins);
    D.0 := ~RD'| MUX(A2: Rx.Q.0, Rx.rdy);
    D.1 := ~RD'| MUX(A2: Rx.Q.1, Tx.rdy);
    FOR i := 2 .. 7 DO D.i := ~RD'|  MUX(A2: Rx.Q.i, '0) END ;
    Tx(~WR'* ~A2, D);
    out := Tx.out
END UART.
```

## 12.4. A Buffered Transmitter and Receiver

A UART as presented in the preceding section sends and receives byte after byte. This implies that between every two consecutive transmissions, interactions with the driving computers on each side have to occur. Each transfer has to be preceded by the computers polling the respective ready signals via their interface. These actions, achieved by programs, i.e. short instruction sequences, reduce the effective transfer rate considerably. This mode of operation is called *programmed I/O.*

*Programmed I/O*

Faster rates are obtained by transmitting longer sequences of bytes, such that the time spent for polling is reduced. The ready signals are polled after transmission of each bit sequence, called a *packet,* rather than after each byte. Subsequently, we will present the design of a transmitter-receiver pair capable of transmitting packets, using a local buffer, a first-in-first-out memory (*Fifo*), to store an entire packet. Before each transmission, the driving computer waits until the transmitter is ready. Then the packet is transferred into the local memory, and the transmitter is triggered. The receiving computer waits until the receiver has received the packet. Then the packet is transferred from the buffer into the computer's store, and the operation ends with resetting the receiver's ready flag. The required configuration is shown in Fig. 12.17.

*Packet*

```
Sender:
    REPEAT UNTIL BIT(Tbase+4, 0)          wait until Tx.rdy
    i := 0;
    REPEAT PUT(Tbase, data[i]); INC(i)
    UNTIL i = paklen;                     load buffer
    PUT(Tbase+4, 0)                       trigger transmitter

Receiver:
    REPEAT UNTIL BIT(Rbase+4, 0)          wait until Rx.rdy
    i := 0;
    REPEAT SYSTEM.GET(Rbase, data[i]);    fetch received bytes
       SYSTEM.PUT(Rbase, 0); INC(i)       increment Adr-counter
    UNTIL i = paklen;
    PUT(Rbase+4, 0)                       reset Rx flag
```

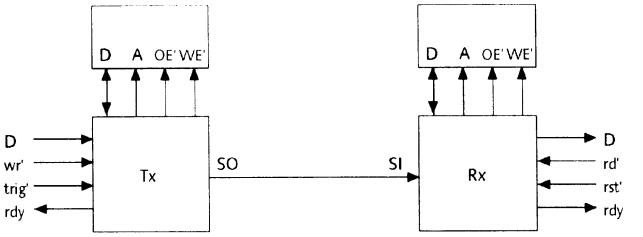



*Fig. 12.17. Transmitter and receiver for packet transmission*

The packet length to be chosen is limited by the accuracy of clock oscillators. The sampling time must not deviate by more than about 1/4 bit time in either direction. An accuracy that is well within the characteristics of crystal oscillators is 0.05%, and it yields a packet length of about 500 bits. We therefore postulate a packet length of 64 bytes, preceded by a start bit and followed by some stop bits.

The other characteristic that must be chosen before a design is attempted is the transmission rate. Here we aim at 10 Mb/s - the same as Ethernet - requiring a line bandwidth of 5 MHz. This is only half that required by Ethernet and allows the use of reasonably cheap twisted-pair cables. Given this frequency, the bit time is 0.1 µs and the byte time 0.8 µs. This allows us to use a standard SRAM for the local buffer memory instead of a special, fast Fifo-RAM. Let us assume the SRAM's cycle time to be (not more than) 200 ns, which is a value easily met by modern SRAM parts. This assumption forces us to incorporate an explicit address counter into the transmitter and receiver.

The chosen frequency also allows the use of the oversampling technique presented in Sect. 12.3. By selecting an oversampling factor of (only) 8, we obtain a basic clock frequency of 80 MHz, which is still acceptable by fast parts. With due regard to the high frequency, the transmitter's prescaler P, dividing the 80 MHz by 8 to obtain the bit clock, is not implemented as a counter, but rather as a (free-running) circular shift register. This solution eliminates combinational circuitry between the elements, and in particular a carry chain with substantial path length.

The obvious choice for the state machine is again the binary counter C. Note that the clock rate for all circuits except the prescaler is the relatively unproblematic 10 MHz. The counter value represents the

number of the bit being transmitted. The state machine must generate the following signals:

| | |
|---|---|
| *zero* | indicating the start bit (and the idle state), |
| *ld* | for loading the shift register Q with the next byte |
| *last* | indicating the last bit (including stop bits) |

*ld* is also used to increment the address counter through its trailing edge. *ld* is active whenever the counter value modulo 8 is 0, except for the last byte period. The timing of these signals is shown in Fig. 12.18. The interface between the computer and the transmitter follows the pattern of Sect. 11.2 with an SR-latch called *flg*. It is set by the external trigger and reset by *last*. A synchronizer lies between the latch and the state machine, generating the signal *run*. *last* also clears the counter upon the next clock cycle. At the same time, *run* goes low, and therefore the counter stops with value 0.

```
flg := SR(trig', ~last);
run := REG(flag * trig');   (*synchronizer*)
ld := ~C.q.0 * ~C.q.1 * ~C.q.2 * ~C.q.9;   (*8, 16, 24, ..., 504 *)
zero := ~C.q.0 * ~C.q.1 * ~C.q.2 * ~C.q.3 * ~C.q.4 * ~C.q.5 *
    ~C.q.6 * ~C.q.7 * ~C.q.8 * ~C.q.9;
last := REG(C.q.0 * C.q.1 * C.q.9);   (*516*)
out := Q.0 * ~zero + ~run;
```

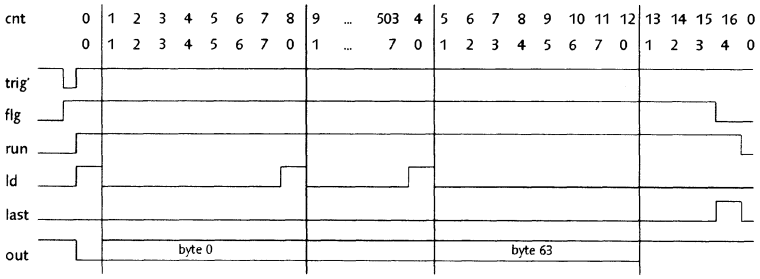



*Fig. 12.18. Transmitter signals*

The state counter C runs from 0 to 516, and consequently requires 10 elements. For reasons becoming clear from the design of the corresponding receiver, we choose 4 stop bits. Their value (1) results not

from logic gates, but simply because 1's are shifted into the shift register when a load signal is missing, as is the case for the "byte time" 64. Signal *last* is used to reset the flag and to turn off the counter. As SR-latches should preferably not be driven by signals that may contain hazards, we do not derive this signal directly by decoding counter values. Instead, a signal is decoded which is active one cycle earlier, and this signal is delayed by a register. Hence, the register output is both active during the desired cycle and free of hazards.

The time for reading the next byte from the buffer after the address counter had been incremented is (almost) 800 ns. This is well within the capabilities of any modern SRAM part. The transmitter circuit is presented in Fig. 12.19 and is based on the timing considerations shown in Fig. 12.18. A complete Lola specification is given at the end of this section.

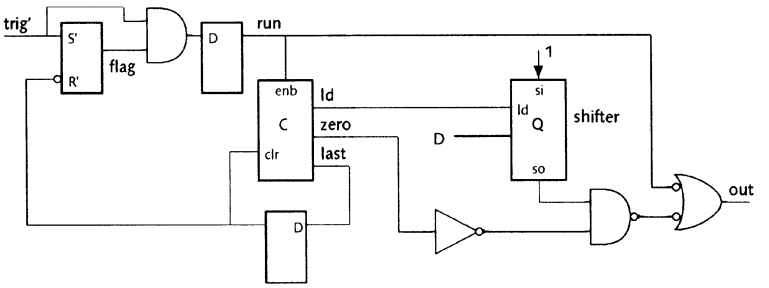



*Fig. 12.19. Transmitter schema*

The design of a receiver follows the pattern given in the preceding section. The differences lie in an extended state machine, implemented as a 10-bit counter C, assuming the role of the UART's bit counter B, and in the UART's clock counter C being replaced by a circular 8-bit shift register P. Its number of elements is given by the oversampling rate. P can be reset, because an *and* gate is placed between any two consecutive elements.

The receiver signals' timing is shown in Fig. 12.20. The prescaler P is enabled by either the counter value being nonzero (transmission proceeding), or by the serial input being 0, i.e. by the start bit of an incoming packet. The rising edge of the clock of counter C arrives 4 prescaler cycles, i.e. 1/2 bit time, after the start bit had been sensed.

The line is consequently sampled in the middle of every bit cell, as is optimal for transmission reliability.

```
zb := ~C.q.3 * ~C.q.4 * ~C.q.5 * ~C.q.6 * ~C.q.7 * ~C.q.8 * ~C.q.9;
zero := ~C.q.0 * ~C.q.1 * ~C.q.2 * zb;
run := ~in + ~zero;
ld := C.q.0 * ~C.q.1 * ~C.q.2 * ~zb;
last := REG(C.q.9 * ~C.q.0 * C.q.1);  (*515*)
we' := ~REG(C.q.1 - C.q.0) * ~C.q.2 * ~zb)
```

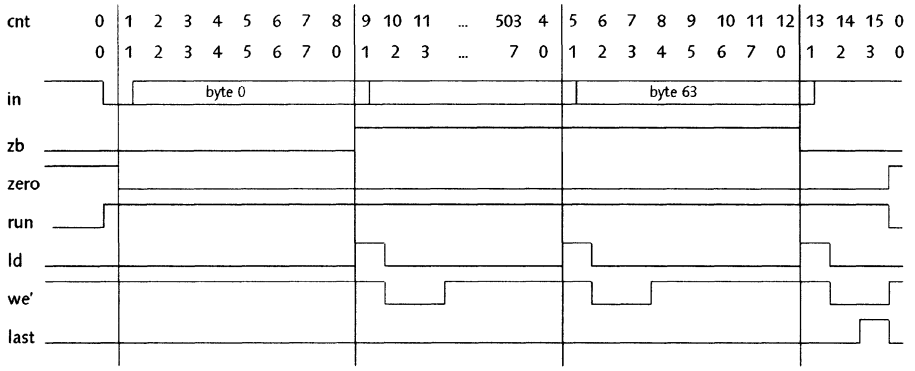



*Fig. 12.20. Receiver signals*

At this point we must mention a memory timing problem. The load pulse can unfortunately not be used to write the data accumulated in the shift register Q directly into the memory buffer. The pulse is one bit time (100 ns) long, which is insufficient for the specified SRAM with a 200 ns cycle time. The problem is solved by inserting a fast one-byte buffer between the shifter and the SRAM. The *ld* signal then serves to enable this intermediate, fast buffer *B*, whereas another signal, *we*, must be generated as write-enable for the SRAM. It evidently must *follow* the *ld* signal and be at least two bit times long. *ld* is active when the counter value is 1 (modulo 8), and *we* is active when it is 2 or 3 (modulo 8), with the exception of the first byte time. Signal *last* clears the counter and stops it, because *zero* is going high, and it also sets the *rdy* flag. This flag is reset by a signal from the receiving computer after it had read out the received packet from the SRAM. The resulting receiver is shown schematically in Fig. 12.21.

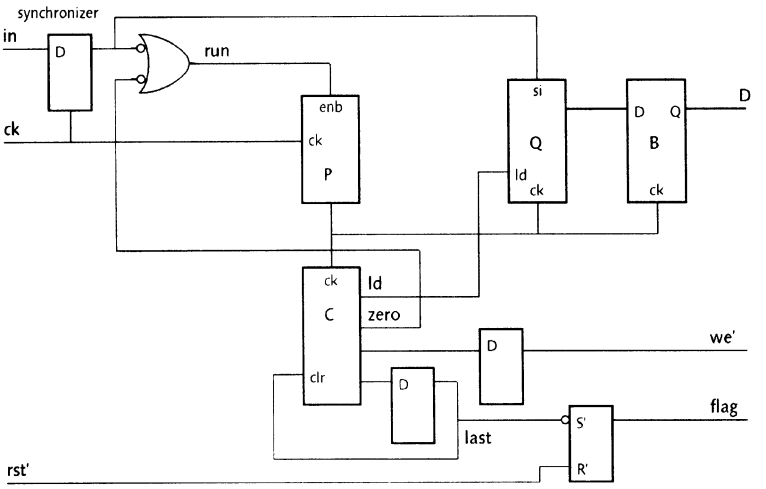



*Fig. 12.21. Receiver schema*

Typically, a transmitter and a receiver are paired together in a single, integrated part. Because of the single buffer, they cannot be operated simultaneously. Note that the address counter is activated

1. when sending, i.e. transferring data from the buffer to the transmitter,
2. when transferring data from the buffer to the reading computer,
3. when receiving, i.e. transferring data from the receiver to the buffer,
4. when transferring data from the sending computer to the buffer.

In the following specification of the entire circuit, the address counter A is, for reasons of simplicity, driven by a gated clock; its terms are the transmitter's *ld* signal (case 1), the receiver's *we* signal (case 3), and an external *wr* signalling the transfer of a byte to or from the data bus (cases 2 and 4). And herewith we conclude this last, substantial exercise in digital circuit design.

```
MODULE TX;
  TYPE Counter;
    IN ck, en, clr: BIT;
    OUT q: [10] BIT;
    VAR c, d: [10] BIT;
```

```
BEGIN
   q.0 := REG(ck: d.0 - en); c.0 := d.0 * en; d.0 := q.0 + clr;
   FOR i := 1 .. 9 DO
      q.i := REG(ck: d.i - c[i-1]); c.i := d.i * c[i-1]; d.i := q.i + clr
   END
END Counter;

TYPE Transmitter;
   IN ck, trig': BIT;
      d: [] BIT;
   OUT out, ld, run: BIT;
   VAR zero, last, flag: BIT;
      P: [8] BIT; (*clock generator*)
      C: Counter;
      Q: [8] BIT; (*shift register*)
BEGIN
   flag := SR(trig', ~last);
   run := REG(P.7: flag * trig');   (*synchronizer*)
   ld := ~C.q.0 * ~C.q.1 * ~C.q.2 * ~C.q.9;
   zero := ~C.q.0 * ~C.q.1 * ~C.q.2 * ~C.q.3 * ~C.q.4 * ~C.q.5 *
      ~C.q.6 * ~C.q.7 * ~C.q.8 * ~C.q.9;
   last := REG(P.7: C.q.0 * C.q.1 * C.q.9);
   out := Q.0 * ~zero + ~run;
   P.0 := REG(ck: ~P.7);
   FOR i := 1 .. 3 DO P.i := REG(ck: P[i-1]) END ;
   P.4 := REG(ck: ~P.3);
   FOR i := 5 .. 7 DO P.i := REG(ck: P[i-1]) END ;
   C(P.7, run, last);
   FOR i := 0 .. 6 DO Q.i := REG(P.7: MUX(ld: Q[i+1], d.i)) END ;
   Q.7 := REG(P.7: MUX(ld: '1, d.7))
END Transmitter;

TYPE AdrCounter;
   IN ck: BIT;
   OUT q: [6] BIT;
   VAR c: [6] BIT;
```

```
    BEGIN q.0 := REG(ck: ~q.0); c.0 := q.0;
      FOR i := 1 .. 5 DO
        q.i := REG(ck: q.i - c[i-1]); c.i := q.i * c[i-1]
      END
    END AdrCounter;

    IN RD', WR', A2, CK: BIT; (*I/O control*)
    INOUT D: [8] TS;  (*I/O bus*)
      d: [8] TS;  (*mem bus*)
    OUT WE', OE', CS': BIT;  (*mem control*)
      a: [6] BIT;  (*mem adr*)
      out, ine', oute: BIT; (*serial output*)
    VAR wr0', wr1': BIT;
      Tx: Transmitter;
      A: AdrCounter;
BEGIN wr0' := WR' + A2; wr1' := WR' + ~A2;
    Tx(CK, wr1', d);
    A(MUX(Tx.run: wr0', ~Tx.ld));
    out := Tx.out; oute := '1; ine' := '1;
    FOR i := 0 .. 5 DO a.i := A.q.i END ;
    FOR i := 0 .. 7 DO d.i := ~wr0' | D.i END ;
    D.0 := ~RD' | MUX(A2: d.0, Tx.run);
    FOR i := 1 .. 7 DO D.i := ~RD' | MUX(A2: d.i, '1) END ;
    WE' := wr0'; OE' := ~wr0'; CS' := '0;
END TX.

MODULE RX;
    TYPE Counter;
      IN ck, en, clr: BIT;
      OUT q: [10] BIT;
      VAR c, d: [10] BIT;
    BEGIN
      q.0 := REG(ck: d.0 - en); c.0 := d.0 * en; d.0 := q.0 + clr;
      FOR i := 1 .. 9 DO
        q.i := REG(ck: d.i - c[i-1]); c.i := d.i * c[i-1]; d.i := q.i + clr
      END
    END Counter;
```

```
TYPE Receiver;
   IN ck, rst', in: BIT;
   OUT B: [8] BIT; (*buffer register*)
      flag, we': BIT;
   VAR zb, zero, run, ld, last: BIT;
      P: [8] BIT; (*clock generator*)
      C: Counter;
      Q: [8] BIT; (*shift register*)
BEGIN zb := ~C.q.3 * ~C.q.4 * ~C.q.5 * ~C.q.6 * ~C.q.7 *
      ~C.q.8 * ~C.q.9;
   zero := ~C.q.0 * ~C.q.1 * ~C.q.2 * zb;
   run := ~in + ~zero;
   ld := C.q.0 * ~C.q.1 * ~C.q.2 * ~zb;
   last := REG(P.7: C.q.9 * ~C.q.0 * C.q.1);
   we' := ~REG(P.7: (C.q.1 - C.q.0) * ~C.q.2 * ~zb);
   P.0 := REG(ck: MUX(run: '0, ~P.7));
   FOR i := 1 .. 3 DO P.i := REG(ck: MUX(run: '0, P[i-1])) END ;
   P.4 := REG(ck: MUX(run: '0, ~P.3));
   FOR i := 5 .. 7 DO P.i := REG(ck: MUX(run: '0, P[i-1])) END ;
   C(P.7, '1, last);
   FOR i := 0 .. 6 DO Q.i := REG(P.7: Q[i+1]) END ;
   Q.7 := REG(P.7: in);
   FOR i := 0 .. 7 DO B.i := REG(P.7: ld, Q.i) END ;
   flag := SR(~last, rst')
END Receiver;

TYPE AdrCounter;
   IN ck: BIT;
   OUT q: [6] BIT;
   VAR c: [6] BIT;
BEGIN q.0 := REG(ck: ~q.0); c.0 := q.0;
   FOR i := 1 .. 5 DO
      q.i := REG(ck: q.i - c[i-1]); c.i := q.i * c[i-1]
   END
END AdrCounter;
```

```
    IN RD', WR', A2, CK: BIT; (*I/O control*)
       in: BIT; (*serial input*)
    INOUT D: [8] TS; (*I/O bus*)
       d: [8] TS; (*mem bus*)
    OUT WE', OE', CS': BIT; (*mem control*)
       a: [6] BIT; (*mem adr*)
    VAR wr0', wr1', ack: BIT;
       Rx: Receiver;
       A: AdrCounter;
BEGIN wr0' := WR' + A2; wr1' := WR' + ~A2;
    Rx(CK, wr1', in);
    A(wr0' * Rx.we');
    FOR i := 0 .. 5 DO a.i := A.q.i END ;
    FOR i := 0 .. 7 DO d.i := ~Rx.we' | Rx.B.i END ;
    D.0 := ~RD' | MUX(A2: d.0, Rx.flag);
    FOR i := 1.. 7 DO D.i := ~RD' | MUX(A2: d.i, '1) END ;
    OE' := ~Rx.we'; WE' := Rx.we'; CS' := '0)
END RX.
```

**Summary**

The maximum data rate of a transmission line with a bandwidth of n Hz is 2n bit/s. The rate decreases, if an encoding of data and clock is required for synchronous transmission over a single wire. Straight frequency or phase modulation achieves n bits/s. Circuits for digital frequency modulation and demodulation are presented in this chapter.

A simpler method is asynchronous transmission. Data are sent in the form of reasonably short packets, and every packet is preceded by a start bit triggering the receiver's clock. Transmitter and receiver clocks then run independently, and are expected to maintain the same frequency sufficiently precisely while a packet is transmitted. This method is used by the well-established RS-232 Standard, where the packet length is very short (at most 9 data bits).

The chapter ends with the design of a transmitter and a receiver for asynchronous, buffered serial transmission. It constitutes a substantial exercise in digital circuit design.

Appendix 1

# Implementations Based on the Programmable Gate Array AT6002

## 1. The Laboratory

The lectures from which this text originates are accompanied by laboratory exercises. Instead of relying on customary laboratory equipment and on the techniques of wire-wrapping, plugging together, or soldering of parts, we designed an interface card for workstations - which typically serve for programming exercises - containing a field-programmable gate array. This gate array provides a highly flexible „battleground" for all sorts of exercises. Because of the numerous cells available, it allows us to issue much more substantial and challenging exercises than would be possible with conventional laboratory equipment.

The extension card connecting the FPGA with the host computer is accompanied with software tools. A design is entered with the help of a graphical *layout editor.* Its consistency with the specification in the form of a Lola program is checked with the aid of a *Lola compiler* and a *Checker.* The Lola compiler translates the Lola program into an abstract data structure, which serves as a basis not only for the Checker but also for other tools, such as a timing analyzer and a simulator. Part of each assignment is also the development of - usually quite simple - test programs for investigating the actual use of the designed circuits. Thus, the workstation is the integrated tool for designing, implementing, and testing the assigned projects.

*Layout editor*
*Consistency check*

We subsequently present solutions to some of the circuits presented in this book. To understand them, the reader will have to know the structure and functionality of the underlying gate array. This structure is therefore presented here in a fair amount of detail. The solutions provided at the end of this appendix also make it clear that finding a layout for a design constitutes a major part of a development. In contrast to programming, designing a circuit in terms of a textual specification (or of a schema) is not sufficient. Equally impor-

tant is the mapping of the abstract specification onto a set of available, physical components (parts). Such parts must be selected from data books, the size of the necessary circuit board must be determined, the parts must be placed, and finally they must be connected.

If the implementation medium is a (programmable) gate array, the available parts are all the same, namely the array's cells. The mapping of circuit elements onto cells is easiest if the cells offer basic functions as flexibly as possible. FPGAs with complicated cells make this task harder. The placement of the various circuit parts is more difficult than with conventional technology, because the facilities for connecting cells (routing) are fixed and limited. No "additional wires" may be drawn anywhere across the board. Hence, the two steps of *placement* and *routing* are strongly interdependent. They appear as very difficult to the beginner. Experience is indeed a substantial help. The exercise solutions presented here are intended to provide a certain amount of "head start".

*Placement*
*Routing*

## 2. The Structure of the Gate Array

The AT6000 series FPGAs principally consist of a matrix of n * n *blocks*, each block consisting of a matrix of 8 * 8 *cells*. For the AT6002 family member, n = 4, yielding a total of 32 * 32 = 1024 cells (Fig. A1.1). Each cell is directly connectable with its 4 neighbours to the north, west, south, and east. It can also be connected to a bus on any side. More about busses will be said later. At the periphery of the array, special cells, also called *pads* or *I/O cells*, are provided for connecting to external signals, i.e. to the pins of the chip.

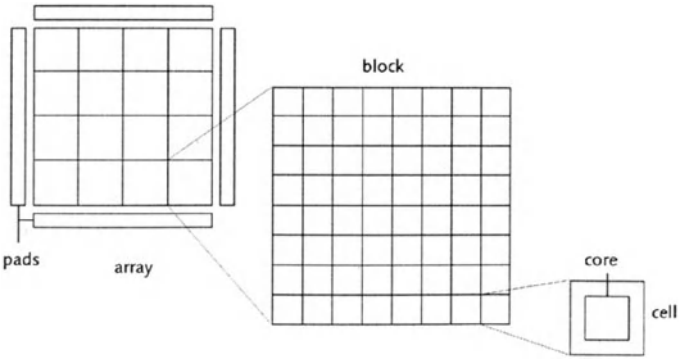



*Fig. A1.1. AT6000 structure*

In explaining the structure of a cell, we start with its *core*. There exist 4 options for its *state*, which determines its functionality. The core has two inputs and two outputs labelled A and B. The 4 possible states are shown in Fig. A1.2. The first two of them serve merely for routing, the third represents a half adder, and the fourth a counter element.

*Cell core*
*State*

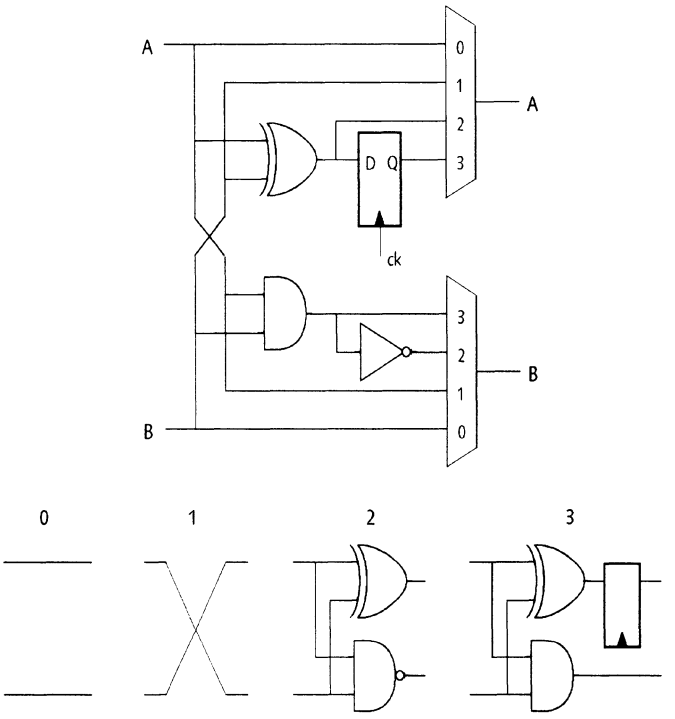



*Fig. A1.2. AT6002 cell core*

The cell's *shell* serves to connect its core with the neighbouring cells and with busses. Outputs A and B of all four neighbouring cells are directly available and selectable as inputs. In addition to the choices from the four directions, the constant 1 is available. Additional gates serve for connecting a bus. On each side of the cell, a bus - a so-called *local bus L*, stretching over a cell block - is reachable. The shell is said to assume a *mode* which determines the bus connections as shown in Fig. A1.3.

*Cell shell*

*Local bus*
*Mode*

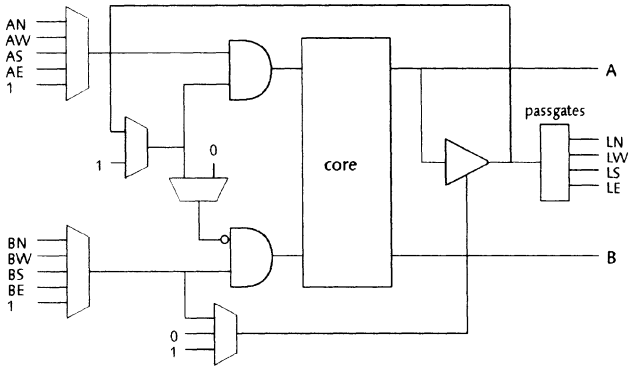



*Fig. A1.3. AT6002 cell shell*

The shell mode essentially offers 4 configurations. They are named according to the function of the bus connection and are shown in Fig. A1.4. (A fifth mode, corner-turn, is explained further below).

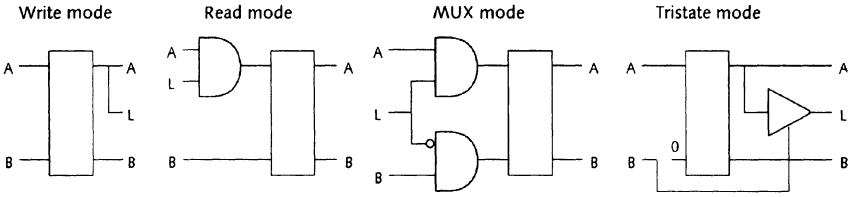



*Fig. A1.4. The four modes of a cell shell*

Cell state and cell mode can be selected independently. Among the resulting, interesting combinations we list the following few:

| mode | state | | A = | B = | L = |
|---|---|---|---|---|---|
| Write | 2 | A = 1 | ~B | ~B | ~B |
| Write | 2 | B = 1 | ~A | ~A | ~A |
| Write | 3 | | REG(A-B) | A*B | REG(A-B) (counter) |
| Mux | 2 | | MUX(L: B, A) | 1 | - |
| Mux | 3 | | REG(MUX(L: B, A)) | 0 | - |

Both outputs of a cell are available as inputs in all four immediate neighbours. In order to lead signals to cells not immediately adjacent, they must either be fed through cells in between which serve as *routing cells* (mode = Write or Read, state = 0 or 1), or they must use busses (see Fig. A1.5).

A (local) bus is provided on each side of a cell, and it leads to all cells in a block lying in the same row or column. At block boundaries, so-called *repeaters* allow bus segments of adjacent blocks to be con-

*Repeater*

nected. These connections are typically one-directional; however, a bidirectional mode for tri-state busses is available. Connections between horizontal and vertical bus segments are possible at every cell through so-called *corner-turns.* In this case, the cell is said to have the *corner-turn mode* which, apart from the bus connections, corresponds to the write-mode. Corner-turns are bidirectional.

*Corner-turn*

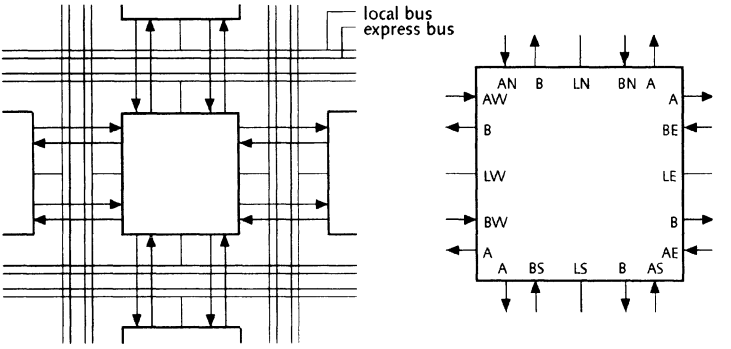



*Fig. A1.5. Cell connections to neighbours and to busses*

Parallel to every local bus segment lies a so-called *express bus* segment. There exist no direct connections to cells, but at either end an express bus can be connected via the block boundary repeater either to the adjacent express bus segment or to its parallel local bus segment. For a detailed description of all repeater and bus options, the reader is referred to the manufacturer's data sheets.

*Express bus*

Each cell contains a D-register. The clock and reset inputs of all cells in the same column are connected by a *clock line* and a *reset line.* The clock line is driven by a multiplexer at the top of the column, the reset line by a multiplexer at the bottom of the column. The multiplexer options are:

*Clock line, reset line*

| clock | reset |
|---|---|
| global clock | global reset (default) |
| lower express bus of top cell row | upper express bus of bottom cell row |
| A output of top cell | A output of bottom cell |
| off (default) | off |

The global clock and reset signals are supplied externally through specific pins.

In order to connect FPGA signals to external signals, special *I/O cells* are provided at the boundary of the cell matrix. Each such cell, whose structure is shown in Fig. A1.6, connects a pin with the A-input of a boundary cell and - via a tri-state driver - with the A-output of an adjacent boundary cell (see Fig. A1.7). On each side, there are connections to 15 pins. The tri-state gate control can be selected as follows:

| | |
|---|---|
| off | the pin is an input |
| on | the pin is an output |
| L-bus parallel to the boundary | the pin is an I/O port |
| L-bus perpendicular to the boundary | the pin is an I/O port |

Note: Actually, there exist 16 I/O cells on each side, one of them being connected to a B-input and B-output. We recommend to ignore this irregularity and to assume 15 cells only.

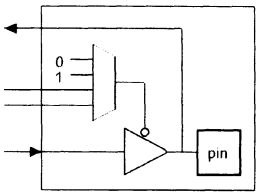



*Fig. A1.6. Structure of I/O cell*

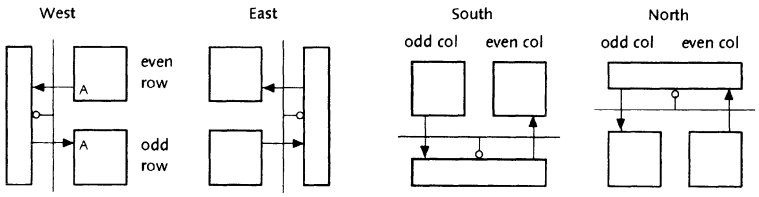



*Fig. A1.7. Connections of I/O cells with boundary cells*

## 3. The FPGA Extension Board

The Programmable Gate Array is connected with the host computer's bus through some interfacing circuitry placed on the extension board. In order to facilitate certain exercises, a few additional parts are also contained on this board. They are an SRAM with a capacity of 32K

bytes, an RS-232 line driver and receiver pair, an RS-485 bus driver and receiver pair, and a crystal oscillator. This configuration is shown schematically in Fig. A1.8.

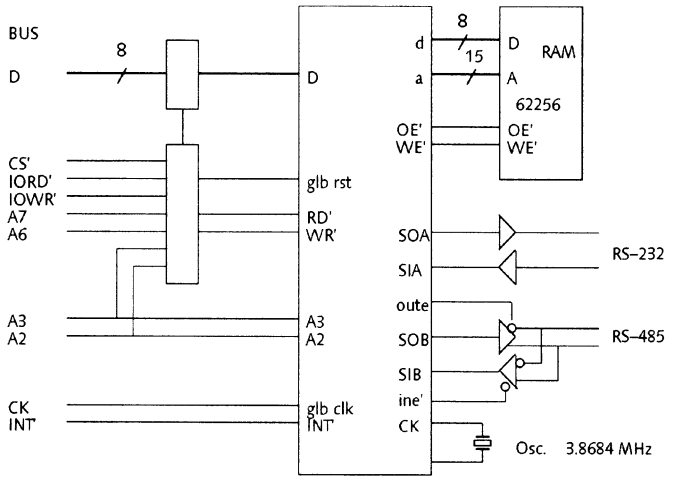



*Fig. A1.8. FPGA extension board*

With the exceptions of the global clock and global reset signals, all indicated signals are available on the FPGA at specific I/O cells. The signal to cell assignment is given in Fig. A1.9 (Connections to special pins used for loading (programming) the FPGA are not shown in Fig. A1.8.)

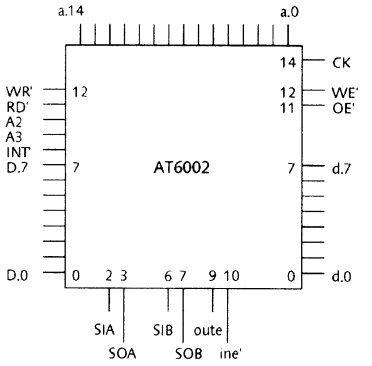
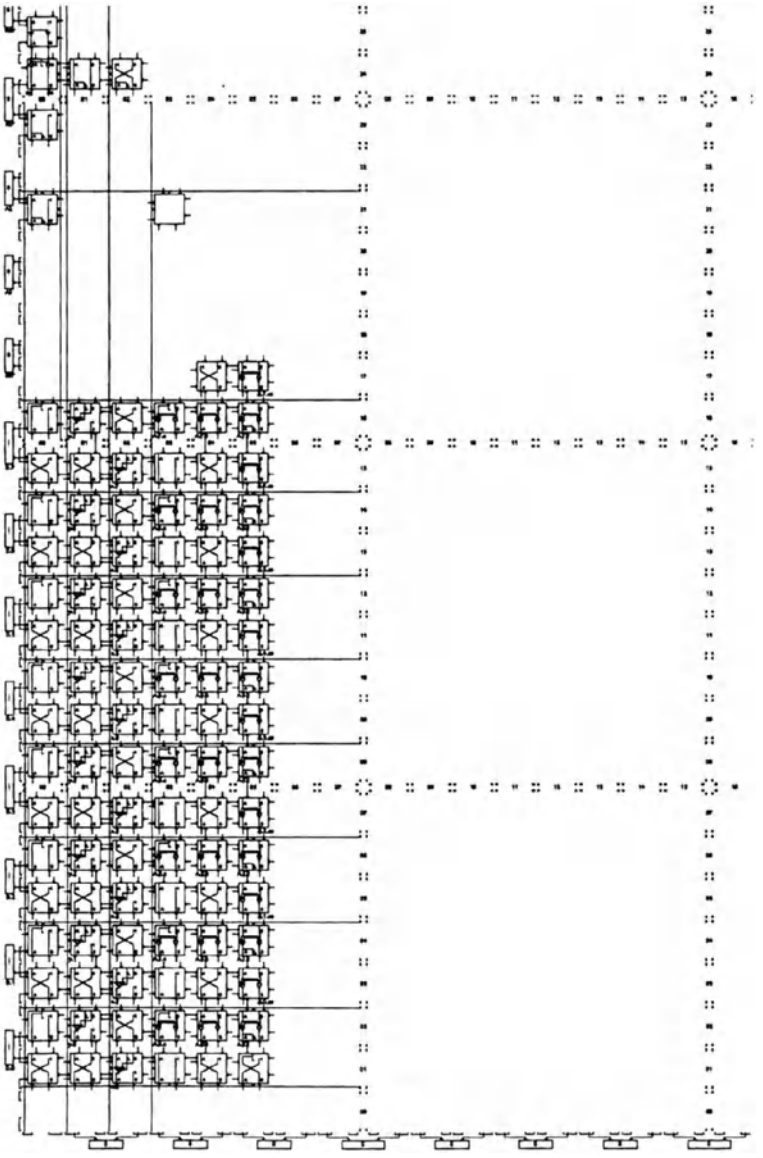


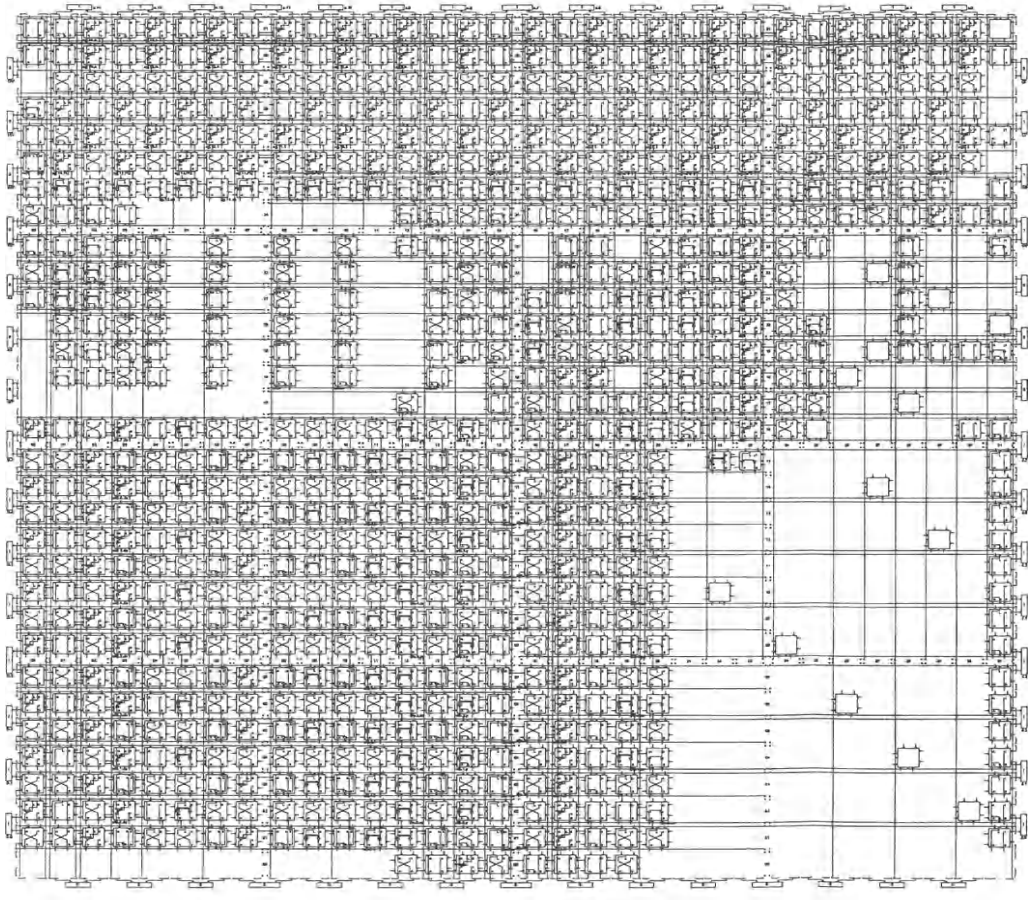

*Fig. A1.9. Signal to I/O cell assignments*

## 4. A Set of Design Examples

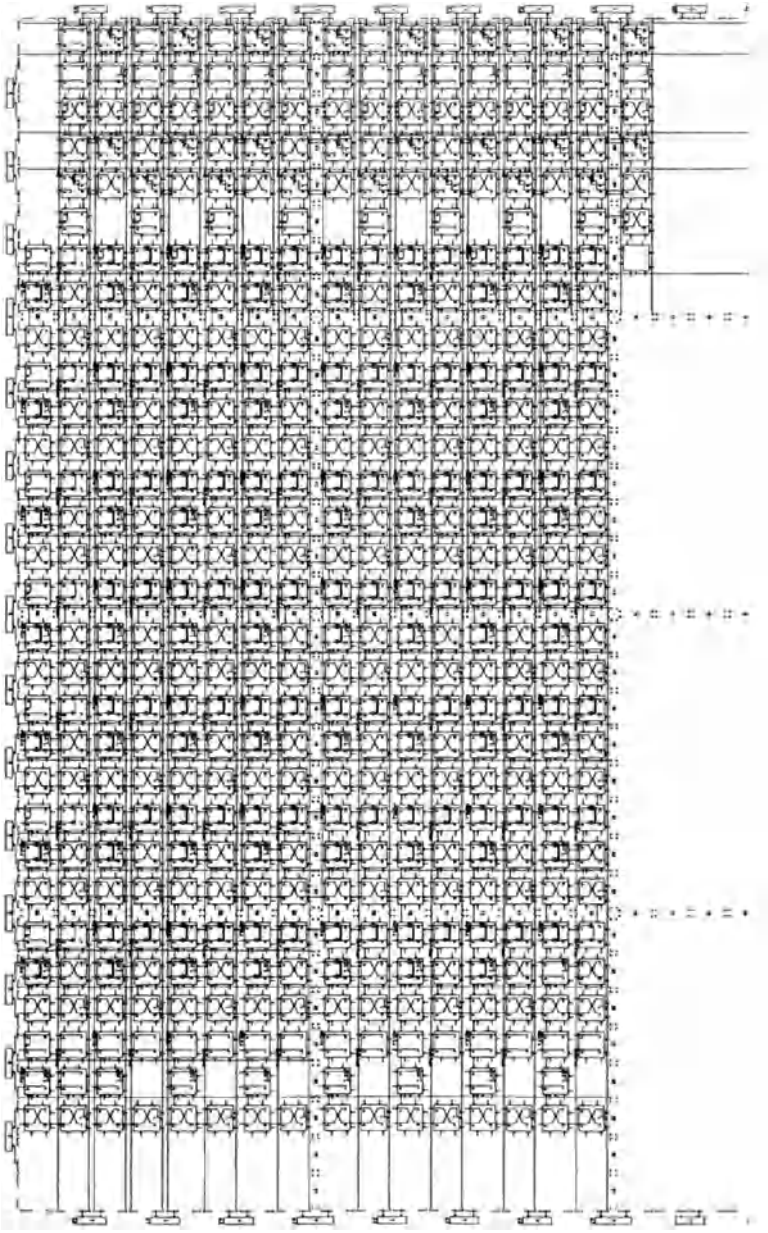
### AddSub.Cli



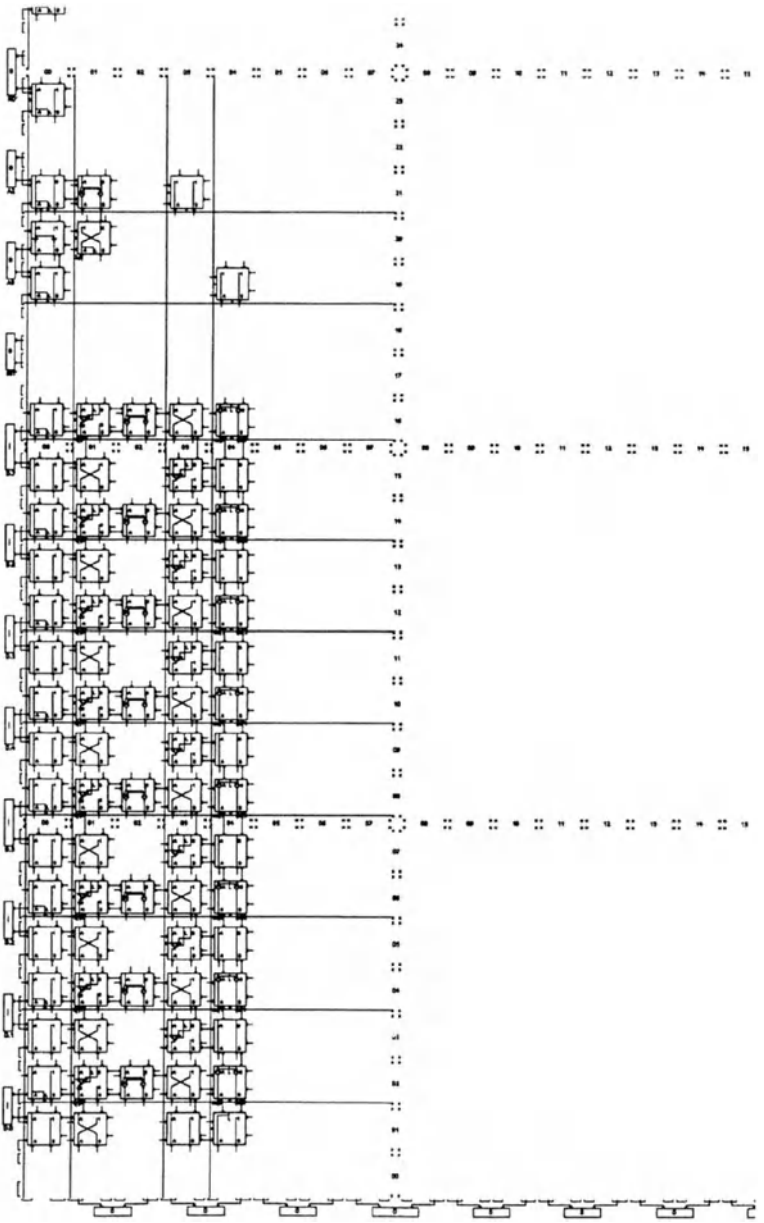
Hercules .CLi



**Multiplier.CLi**



## UDCntr.Cli



Appendix 2

# Syntax of Lola

```
identifier =        letter {letter | digit} [" ' "].
integer =           digit {digit}.
LogicValue =        " '0" | " '1".
BasicType =         "BIT" | "TS" | "OC".
SimpleType =        BasicType | identifier ["(" ExpressionList ")"].
ExpressionList =    expression {"," expression}.
type =              { "[" expression "]" } SimpleType.
ConstDeclaration =  identifier ":="expression ";".
VarDeclaration =    IdList ":" type ";".
IdList =            identifier {"," identifier}.
selector =          {" . " identifier | " . " integer | "[" expression "]"}.
factor =            identifier selector | LogicValue | integer |
                    "~" factor | "(" expression ")" |
                    "MUX" "(" expression ":" expression "," expression ")" |
                    "REG" "(" [expression ":"][expression ","] expression ")" |
                    "LATCH" "(" expression "," expression ")" |
                    "SR" "(" expression "," expression ")" .
term =              factor {("*" | "/" | "DIV" | "MOD" | "↑") factor}.
expression =        term {("+" | "-") term}.
assignment =        identifier selector ":=" [condition "|"] expression.
condition =         expression.
relation =          expression ("=" | "#" | "<" | "<=" | ">" | ">=") expression.
IfStatement =       "IF" relation "THEN" StatementSequence
                    {"ELSIF" relation "THEN" StatementSequence}
                    ["ELSE" StatementSequence]
                    "END" .
ForStatement =      "FOR" identifier ":=" expression ".." expression "DO"
                    StatementSequence "END" .
UnitAssignment =    identifier selector "(" ExpressionList ")".
statement =         [assignment | UnitAssignment | IfStatement | ForStatement].
```

```
StatementSequence = statement {";" statement}.
InType =          {"[" expression "]"} "BIT".
InOutType =       {"[" expression "]"} ("TS" | "OC").
OutType =         {"[" expression "]"} ("BIT" | "TS" | "OC").
ImportList =      "IMPORT" identifier {" , " identifier} ";" .
module =          "MODULE" identifier ";" [ImportList]
                  {TypeDeclaration ";"}
                  ["CONST" {ConstDeclaration}]
                  ["IN" {IdList ":" InType ";"}]
                  ["INOUT" {IdList ":" InOutType ";"}]
                  ["OUT" {IdList ":" OutType ";"}]
                  ["VAR" {VarDeclaration}]
                  ["CLOCK" expression ";"]
                  ["BEGIN" StatementSequence]
                  "END" identifier "." .
TypeDeclaration = "TYPE" identifier ["*"] ["(" IdList ")"] ";"
                  ["CONST" {ConstDeclaration}]
                  ["IN" {IdList ":" InType ";"}]
                  ["INOUT" {IdList ":" InOutType ";"}]
                  ["OUT" {IdList ":" OutType ";"}]
                  ["VAR" {VarDeclaration}]
                  ["BEGIN" [StatementSequence]
                  "END" identifier.
```

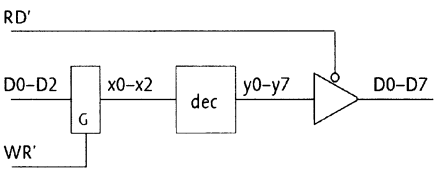
# Selected Design Exercises

Here we present a set of typical design exercises. They are geared to the use of a field programmable gate array as laboratory equipment (see also Appendix 1). Their solution includes the specification of a circuit in terms of Lola and as a schematic diagram, and its implementation as a layout of FPGA cells. For testing the circuit, simple commands for reading and writing data are to be used. In the more extensive exercises, short programs are to be written which test the circuit adequately. The interface of the FPGA to the host computer is assumed to consist of an 8-bit data bus D, 2 address lines (A2 and A3), a read strobe RD', and a write strobe WR'. The strobes are generated by GET and PUT statements respectively.

### Combinational Circuits

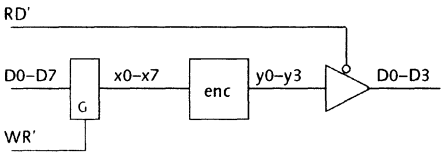
Inputs are held in latches controlled by the write strobe WR'. The read strobe RD' merely controls the tri-state gate connecting the output to the data bus. No clocks are involved, of course.

1. Design and test a 3-to-8 decoder with latched inputs.

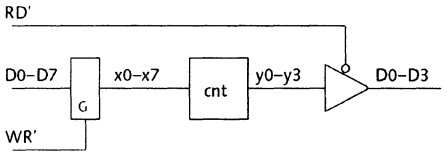



```
MODULE Dec;
   IN RD', WR': BIT;
   INOUT D: [8] TS;
   VAR x: [3] BIT; y: [8] BIT;
BEGIN ...
   FOR i := 0 .. 2 DO x.i := LATCH(~WR', D.i) END;
   FOR i := 0 .. 7 DO D.i := ~RD' | y.i END
END Dec.
```

2. Design and test an 8-to-3 priority encoder with latched inputs. The output y0-y3 is the binary encoded index k, such that d[k] = 1 and d[i] = 0 for all i > k. D3 indicates, whether any input has value 1.




3. Design and test a counter of ones among the latched inputs x0-x7.



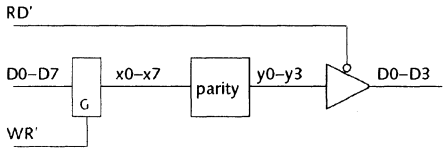

4. Design and test a parity generator for the latched inputs x0-x7 such that

   y0 = x0 - x2 - x4 -x6

   y1 = x1 - x2 - x5 - x6

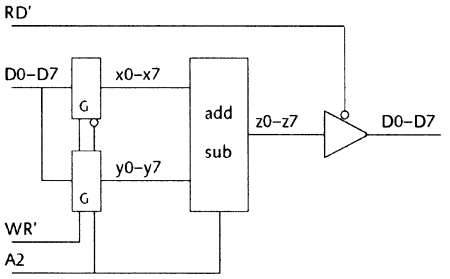
   y2 = x3 - x4 - x5 - x6

   y3 = x7



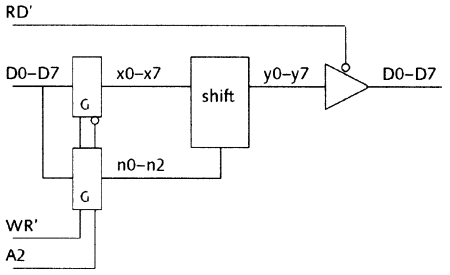

Note that a corresponding parity checker can determine the correct inputs from x0-x7 and y0-y3, if a single signal among them is wrong. Hence, y0-y3 allow not only error checking, but also error correction.

5. Design and test an adder-subtractor. A = 0 shall cause addition z = x+y, A =1 subtraction z = x-y.




6. Design and test an 8-bit comparator. Output z0 signals equality (x = y), and z1 indicates the relation x < y. Assume that the comparands are represented in 2's-complement form.

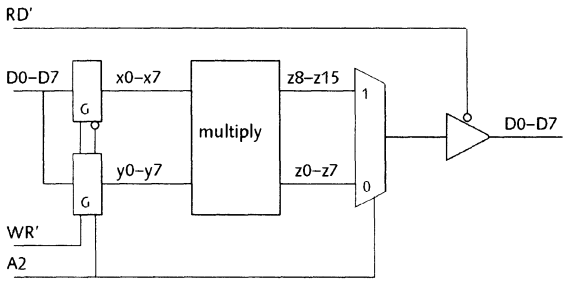
7. Design and test an 8-bit barrel shifter such that y[i] = x[(i+n) MOD 8] for i = 0 .. 7.




Extend your design, such that A2 = 0 signifies rotation of x as before, whereas A2 = 1 signifies a right shift:

y[i] = x[i+n] for i = 0 .. 7-n
y[i] = 0 for i = 8-n .. 7

8. Design and test an 8-bit by 8-bit binary multiplier for 2's-complement numbers. A2 = 0 is to yield the lower, A2 = 1 the higher 8 bits of the 16-bit product z = x*y.




### Sequential Circuits

Testing of circuits typically occurs in the single-step mode. The common clock signal for all registers is the write strobe WR'. This fact is expressed in the short Lola programs by the statement *CLOCK WR'*.

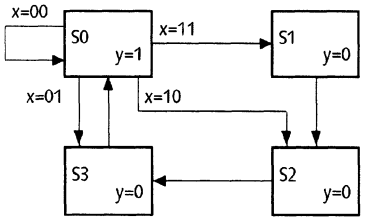
1. Design and test a state-machine which acts as a frequency divider (of the clock signal). The latched input determines the dividing factor:

   d = 0 0 $F_y = F_{ck}/3$

   d = 0 1 $F_y = F_{ck}/5$

   d = 1 1 $F_y = F_{ck}/7$

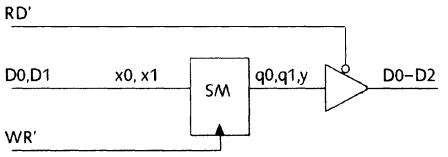
2. A state machine with inputs x1 and x0 and output y is specified by the following state diagram:



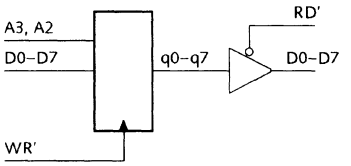

   a. Design and test a one-hot solution. Testing is to be achieved by a single-step operation, where the clock pulse is generated by a write-strobe from the host computer.

```
MODULE SM;
   IN RD', WR': BIT;
   INOUT D: [4] TS;
   VAR x0, x1, y: BIT; q: [4] BIT;
   CLOCK WR';
BEGIN x0 := D.0; x1 := D.1; ...
   FOR i := 0 .. 3 DO D.i := ~RD' | q.i END
END SM.
```

b. Find a suitable encoding of the states and a solution using 2 registers only. Derive a transition table and from it the expressions for the register inputs and the output.




3. Design and test an 8-bit up/down counter with an explicit hold and load function.

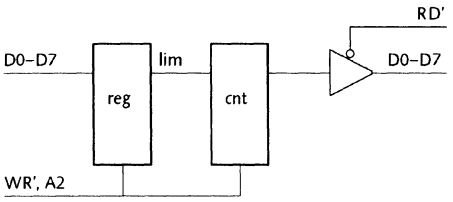



A3,A2 = 0 0 hold
0 1 load
1 0 count up
1 1 count down

```
MODULE UDCnt;
   IN RD', WR', A2, A3: BIT;
   INOUT D: [8] TS;
   VAR q: [8] BIT;
   CLOCK WR';
BEGIN ...
   FOR i := 0 .. 7 DO D.i := ~RD' | q.i END
END UDCnt.
```

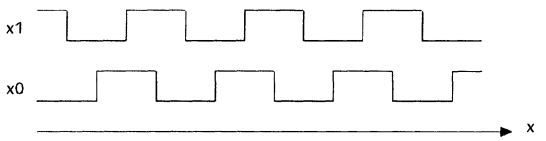
4. Design and test a 2-digit decimal counter. Each of the 2 counter elements represents a binary-coded decimal digit (BCD) consisting of 4 registers. Effectively, it differs from a binary 4-bit counter only by the fact that state "9" instead of state "15" is followed by state "0".

5. Design and test a counter which counts from 0 to a limit which is held in a loadable register.

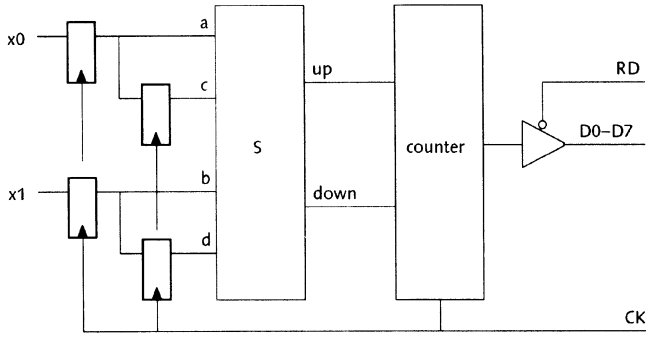



A2 = 0 load limit register and reset counter
A2 = 1 count up (modulo limit+1)

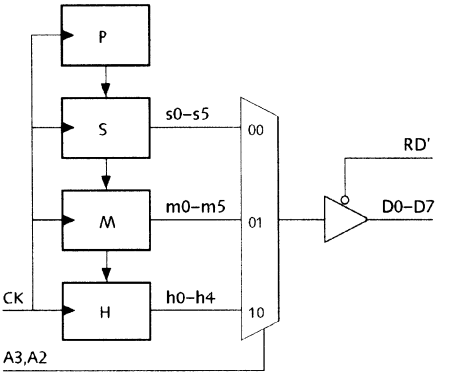
6. Design and test a circuit which constitutes an interface between a mouse and a computer. It consists of two identical parts for the x- and y-coordinates. Let us consider the part for the x-coordinate. There are two inputs from the mouse, called x0 and x1, originating from a sensor of the x-position:




The circuit consists of three parts. In the first, signals x0 and x1 are sampled at a fixed and sufficiently high rate and latched in registers. In the second part, a purely combinational circuit, two signals *up* and *down* are generated. They serve as enable inputs to the third part, which is an up/down counter representing the mouse position. *up* has the value 1, if a movement to the right has been sensed during the last clock period (i.e. if $a \neq c$ or $b \neq d$). *down* is 1, if a movement to the left had been sensed. The entire circuit has the following structure:




7. Design and test a synchronous circuit representing a clock. The outputs are s0 - s5 (seconds), m0 - m5 (minutes), and h0 - h4 (hours). The input ciock signal originates from a crystal oscillator with a frequency of 32768 Hz.




**Memories**

1. Design and test a circuit representing a stack. It consists of m * n registers (or latches), where m is the stack's depth and n its width. Choose, e.g. m = 5 and n = 8. The controls are A2 and the strobes RD' and WR'. Storing a value x on the stack is achieved by applying x to the bus D and 11 to A3, A2, and then issuing a write-strobe. Fetching an element from the stack is achieved by reading the bus D while applying a read-strobe, and by subsequently issuing a write-strobe with address A3, A2 = 10 for adjusting the address pointer. A3 acts as an enable signal for the stack.

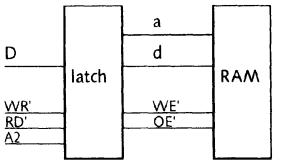
```
MODULE Stack;
   IN RD', WR', A2, A3: BIT;
   INOUT D: [8] TS;
   VAR q: [8][5] BIT;
   CLOCK WR';
BEGIN ...
   FOR i := 0 .. 7 DO D.i := ~RD' | q.i.0 END
END Stack.
```

Design a solution using a gated clock (WR' + ~A3), and a solution without gated clock. Compare the complexity of the two circuits obtained.

In the following exercises we assume that the FPGA is connected to a static RAM by the data bus d, the address lines a, the RAM-output enable signal OE', and the write enable signal WE'.

2. Design and test an address latch connected with a static RAM according to the following specification:
   write, A2 = 0: latch address D
   write, A2 = 1: store data D in RAM at previously latched address
   read: read data D from RAM at previously latched address

```
MODULE RAM;
   IN RD', WR', A2: BIT; (*read/write strobes from computer*)
   INOUT D: [8] TS; (*computer data bus*)
         d: [8] TS; (*memory data bus*)
   OUT WE', OE': BIT; (*memory controls*)
         a: [8] BIT; (*memory address*)
   VAR wr0, wr1: BIT;
BEGIN wr0 := ~WR' +~A2; wr1 : ~WR' + A2;
   WE':= ~wr1; OE' := RD';
   FOR i := 0 .. 7 DO
         a.i := LATCH(wr0, D.i); d.i := wr1 | D.i;
         D.i := ~RD' d.i
   END
END RAM.
```




3. In your design of exercise 2, replace the address latch by a counter. Its dimension depends on the size of the available RAM. The following operations are to be implemented:

| | | |
|---|---|---|
| write, A3, A2= | 00: | load low 8 bits of counter with data D |
| write | 01: | load high 8 bits of counter with data D |
| write | 10: | increment counter |
| write | 11: | write data D into RAM at address held in counter, then increment counter |
| read | | read data D from RAM at address held in counter |

Evidently, it is now possible to store a sequence of bytes at consecutive addresses without supplying an explicit address for each byte. Note that a read-strobe has no effect on the counter.

4. Add a second address counter to your design of exersise 3 and implement the following operations:

| | | |
|---|---|---|
| write, A3, A2= | 00: | no action |
| write | 01: | increment R-counter |
| write | 10: | write data D into RAM at address in W-counter, increment W-counter |
| read, A2 = | 0: | read data D from RAM at address held in counter R-counter |
| read | 1: | D.0 indicates whether R-counter = W-counter |

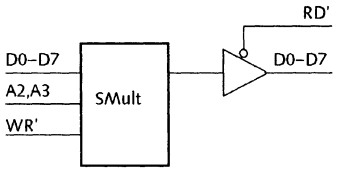
These operations represent a fifo-memory (first-in-first out). The status bit *eql* (to be read with A2 = 1 ) indicates whether the two counters have equal values, in which case the fifo is empty (or full). Suggest a simple measure to distinguish between empty and full.

### Hercules

1. Extend the design of the Hercules computer presented in Chapter 8 by providing a second accumulator register B. The instruction format and encoding must be changed accordingly to allow for a register specification (A or B).

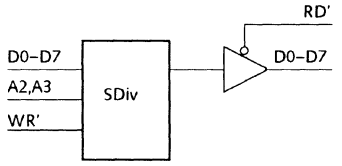
### Multiplication and Division

1. Design and test a serial multiplier circuit according to the scheme presented in Sect. 9.3.




| | | |
|---|---|---|
| write, A3,A2 = | 0 0: | load x from D, hold y, clear z |
| write | 0 1: | hold x, load y from D, clear z |
| write | 1 0: | multiply step |
| read, A2 = | 0: | read x (low part of product) |
| read | 1: | read z (high part of product) |

2. Design and test a serial divider circuit according to the scheme presented in Sect. 9.3.

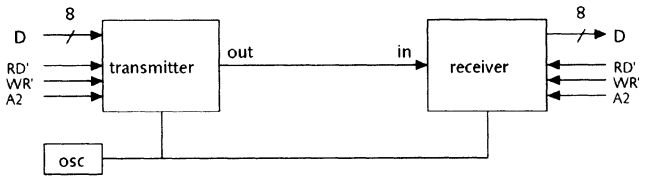



| | | |
|---|---|---|
| write, A3,A2 = | 0 0: | load q from D, hold y, clear r |
| write | 0 1: | hold q, load y from D, clear r |
| write | 1 0: | divide step |
| read, A2 = | 0: | read q (quotient) |
| read | 1: | read r (remainder) |

**Serial DataTransmission**

1. Design and test a transmitter circuit and a receiver circuit for the transmission of packets with the following format:

   1 start bit, value 0
   8 data bits (D0 - D7)
   2 stop bits, value 1




Both parts operate with the same clock signal, generated by an oscillator with a frequency between 1 and 10 MHz. For initial testing, it is recommended to replace the oscillator by a write-strobe from the host computer to achieve single-step operation.

| | | |
|---|---|---|
| write | A2 = 0 | latch data D and trigger transmitter |
| write | A2 = 1 | clock pulse for all registers (single step mode) |
| read | A2 = 0 | read received data |
| read | A2 = 1 | D.0 = Rx.rdy (data received) |
| | | D.1 = Tx.rdy (data sent) |

2. Extend the design of a UART presented in Sect. 12.3 by a parity generator in the transmitter and a parity checker in the receiver. The packet is extended by a parity bit following the 8 data bits.

3. Extend the design of a buffered packet transmission circuit presented in Sect. 12.4 by an error detection scheme. A cyclic redundancy check sum (CRC) is to be generated in the transmitter and checked in the receiver. The packet format is extended by appending the CRC to the 512 data bits. The CRC consists of 8 bits.

4. Design and test a transmitter and a receiver circuit for transmission of packets of fixed lengths similar to the scheme presented in Sect. 12.4, but using synchronous instead of asynchronous transmission. Use frequency modulation as presented in Sect. 12.2. Experiment with different packet lengths.

# Index

# Springer-Verlag and the Environment

We at Springer-Verlag firmly believe that an international science publisher has a special obligation to the environment, and our corporate policies consistently reflect this conviction.

We also expect our business partners – paper mills, printers, packaging manufacturers, etc. – to commit themselves to using environmentally friendly materials and production processes.

The paper in this book is made from low- or no-chlorine pulp and is acid free, in conformance with international standards for paper permanency.